A Concise and OPINIONATED *History* of Virtual Machines

Mario Wolczko
Architect, Oracle Labs

The following is provided for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Overview

Goal: introduce the most important ideas and systems in VM design and implementation from a historical perspective

Audience: I am assuming the typical user's understanding of VM internals (i.e., not much).

Limitations: can't be detailed, or anywhere near complete; deliberately omitting GC, interpretation and compiler history (outside of VMs)

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

CS294-113
A Shameless Plug

- In 2015 I was invited to teach a graduate course on VMs at UC Berkeley.
- The result is CS294-113,Virtual Machines and Managed Runtimes.
- Available on the web (slides, video, exercises) at www.wolczko.com/CS294.
- ~30 hours of video, over 1200 slides. Estimated 200+ hours to complete coursework.
- Guest appearances: Deutsch & Schiffman, Ungar, Click, Bak, Bolz, Würthinger, Van De Vanter
- Taken and completed by >20 UCB students. Nobody dropped out!

What do I mean by “history”?

- I am not a professional historian
- I don’t even play one on TV
- I am more a participant and witness than a chronicler
- Hence, this is a subjective history
- It is laced with my opinions — some of which ~~may be~~ are wrong.

Who am I?

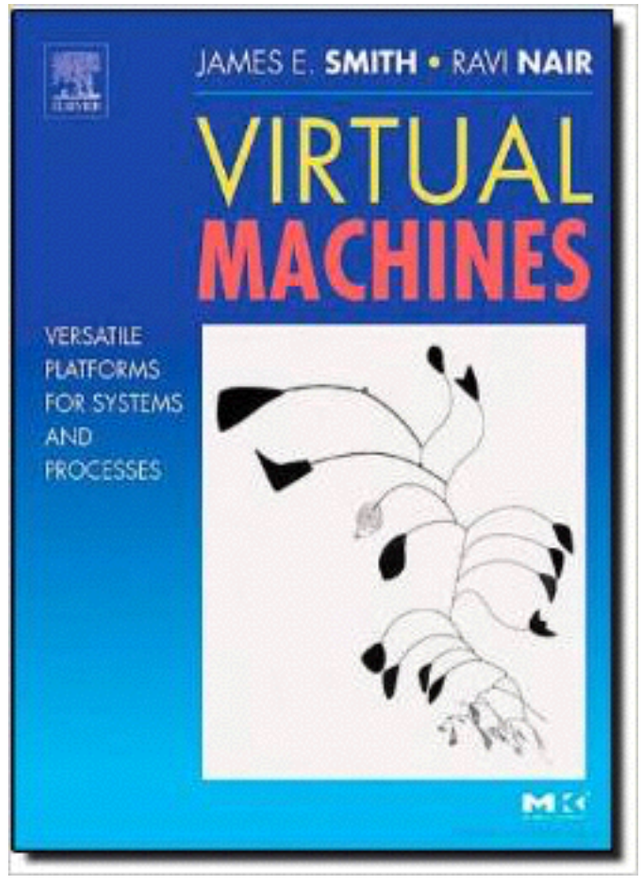
- Architect at Oracle, formerly a Distinguished Engineer at Sun.
- I've been mostly in research, with occasional forays into product development.
- I started my career in VMs at the beginning of a golden era (1983), and have seen many developments up close (esp. Java, at Sun).
- I've been fortunate to work with and talk to many VM pioneers.

1. Introduction to VMs

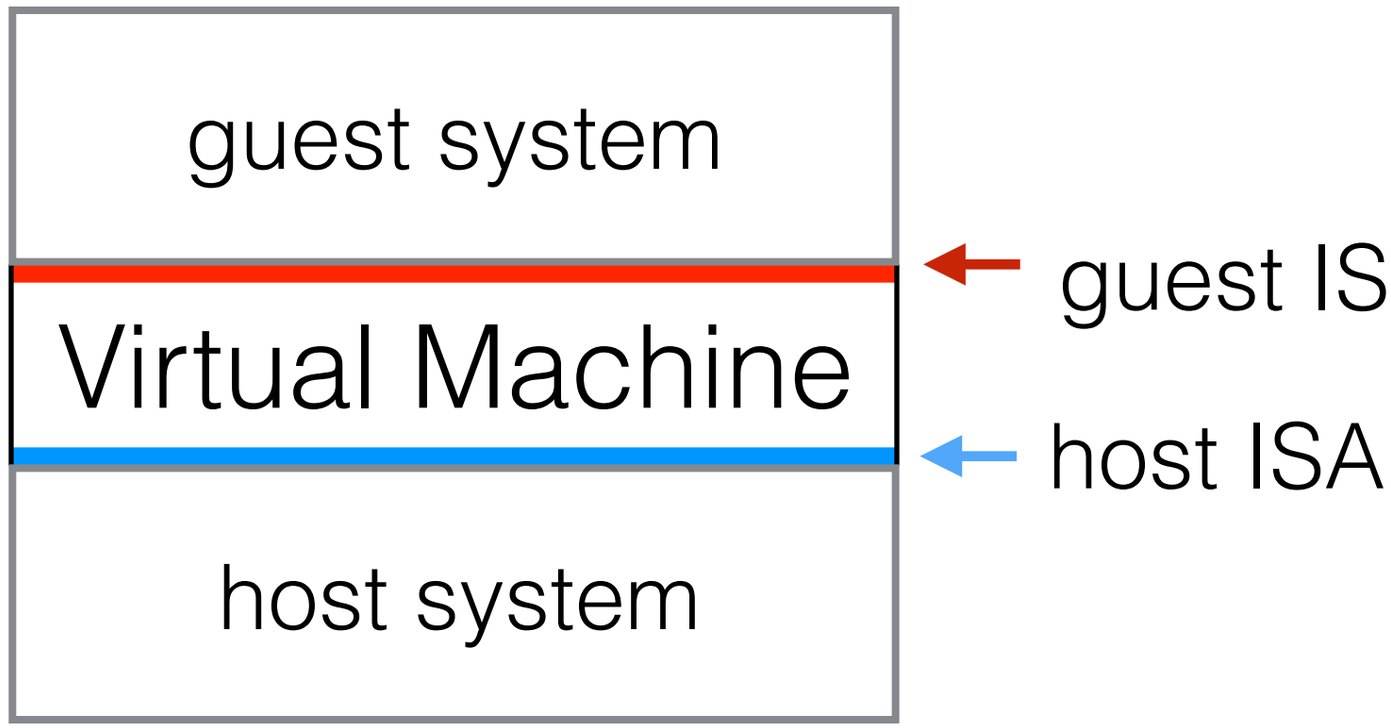
Background reading

- Smith and Nair, *Virtual Machines*, 2005
  I borrow some of their terminology and notation from Chapter 1

- I will introduce papers and other books when appropriate



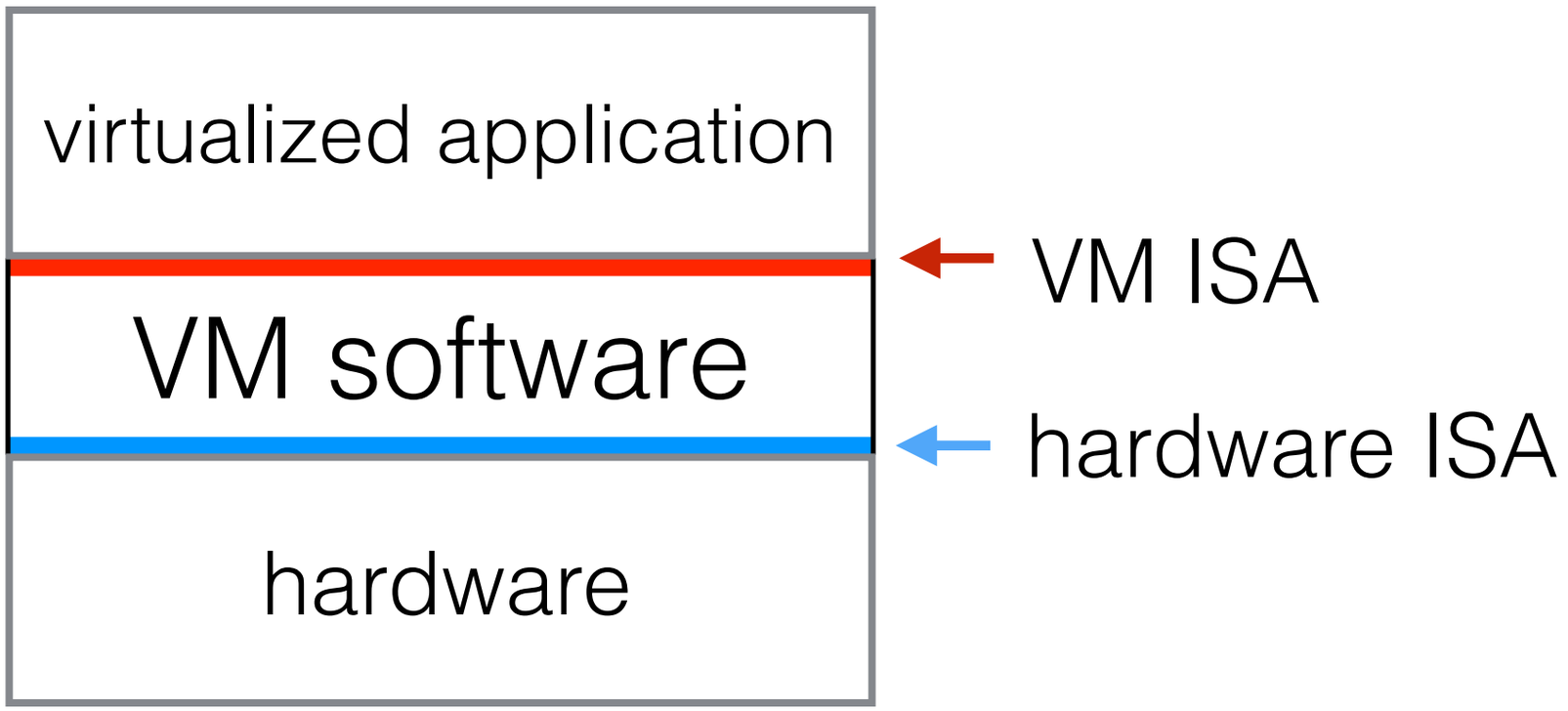
Generic VM architecture



What is a Virtual Machine?

- A software implementation of a machine architecture
  - Software needs hardware to run, so hardware is implied too
- Two machine architectures are involved: *guest* and *host* (although they might be the same! — more later)
- The guest may be defined only by software or be an emulation of a real machine (i.e., also available as a hardware implementation)
- The host is usually hardware, but need not be (e.g., a language VM written in Java running on the JVM)

A typical VM


virtualized application
VM software
hardware
VM ISA
hardware ISA

Process and System VMs

- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)

- A **System VM** implements *both* the hardware user and system ISA

Language VMs

- A language-specific process VM
- The VM presents an OS-like interface to applications in the chosen language, as well as an ISA designed specifically for the semantics of the chosen language.

The timeline

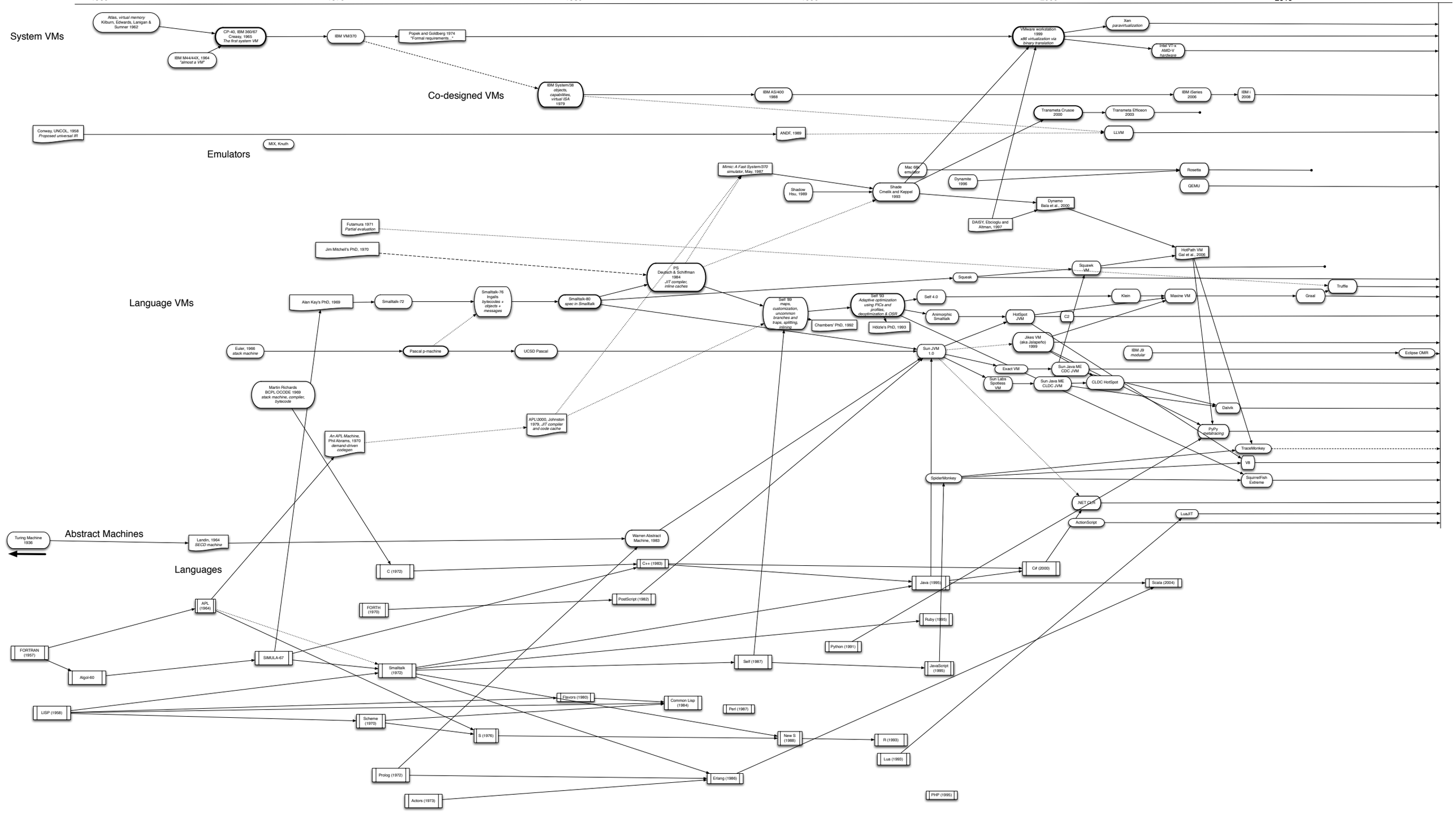
- A timeline of selected landmarks (systems, publications) in VM history

- I'll refer to a printed copy

- Systems and publications in the top ⅔, grouped by category.

- Languages in the bottom ⅓ are for context

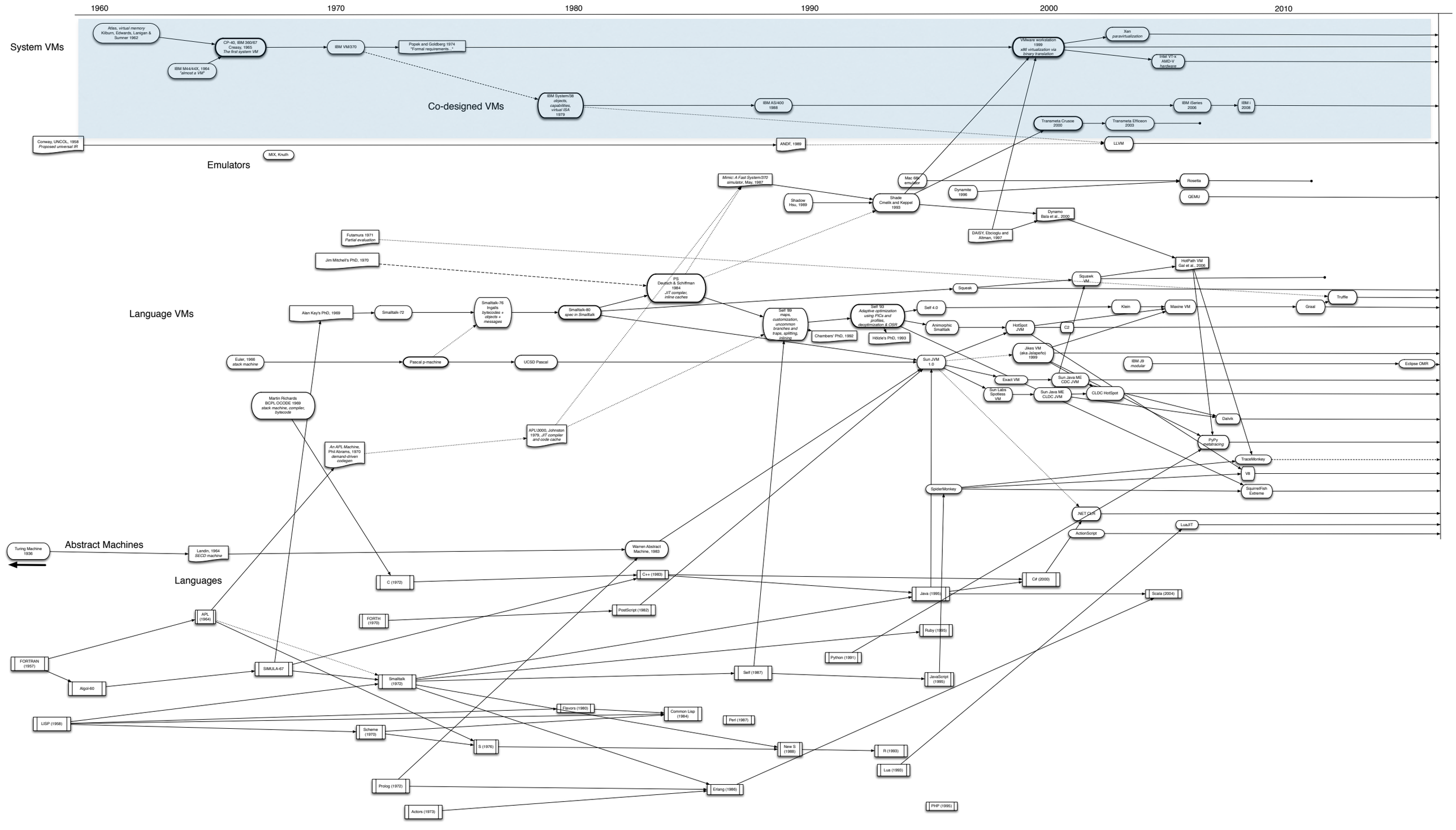
- *Not* exhaustive — "selected" to make a point. Feel free to argue with the choices.



omitted: GC, compilers, interpreters

A Timeline of Selected Landmarks in Virtual Machine History

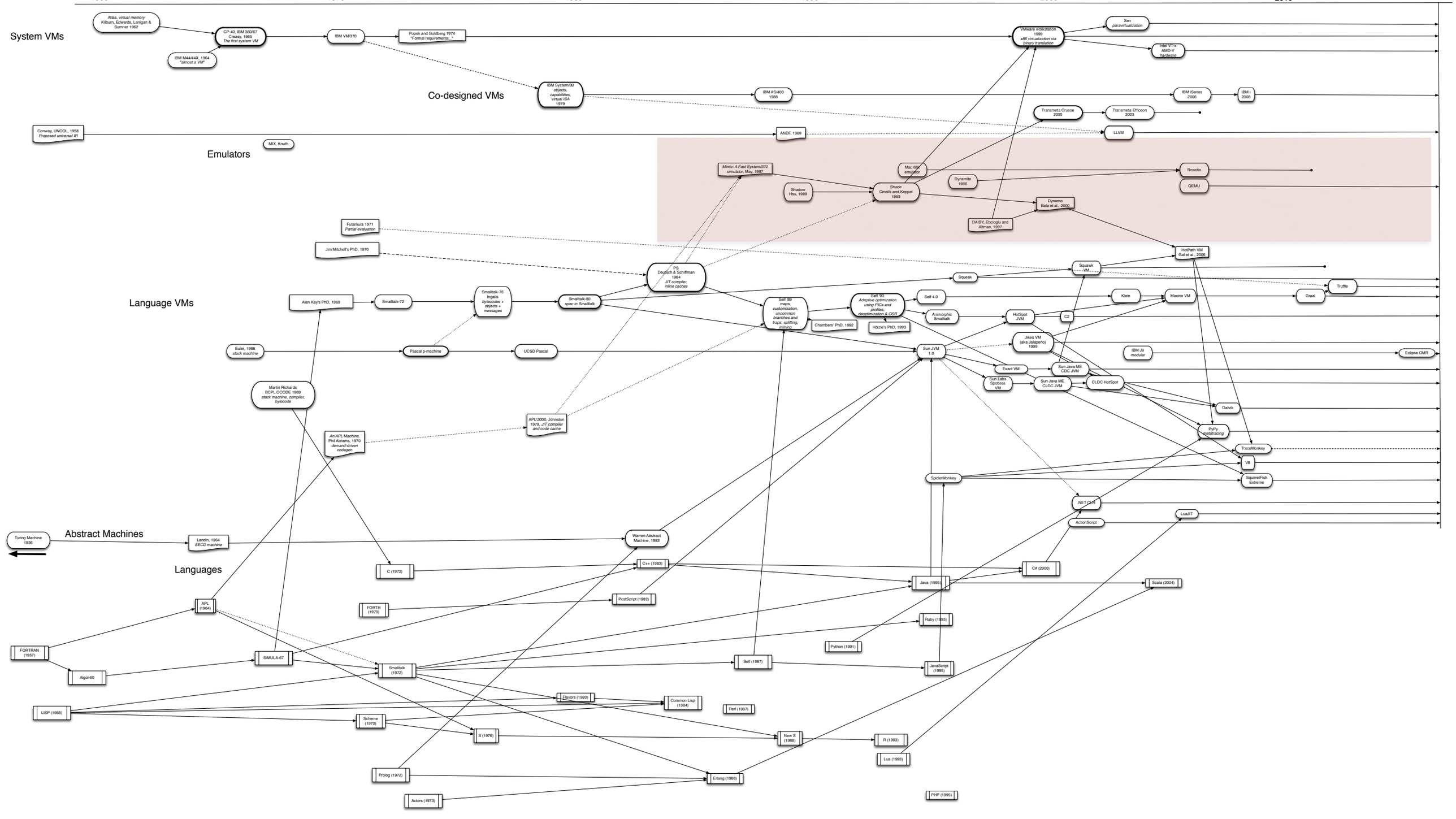
0.99 2016-05-20



omitted: GC, compilers, interpreters

A Timeline of Selected Landmarks in Virtual Machine History

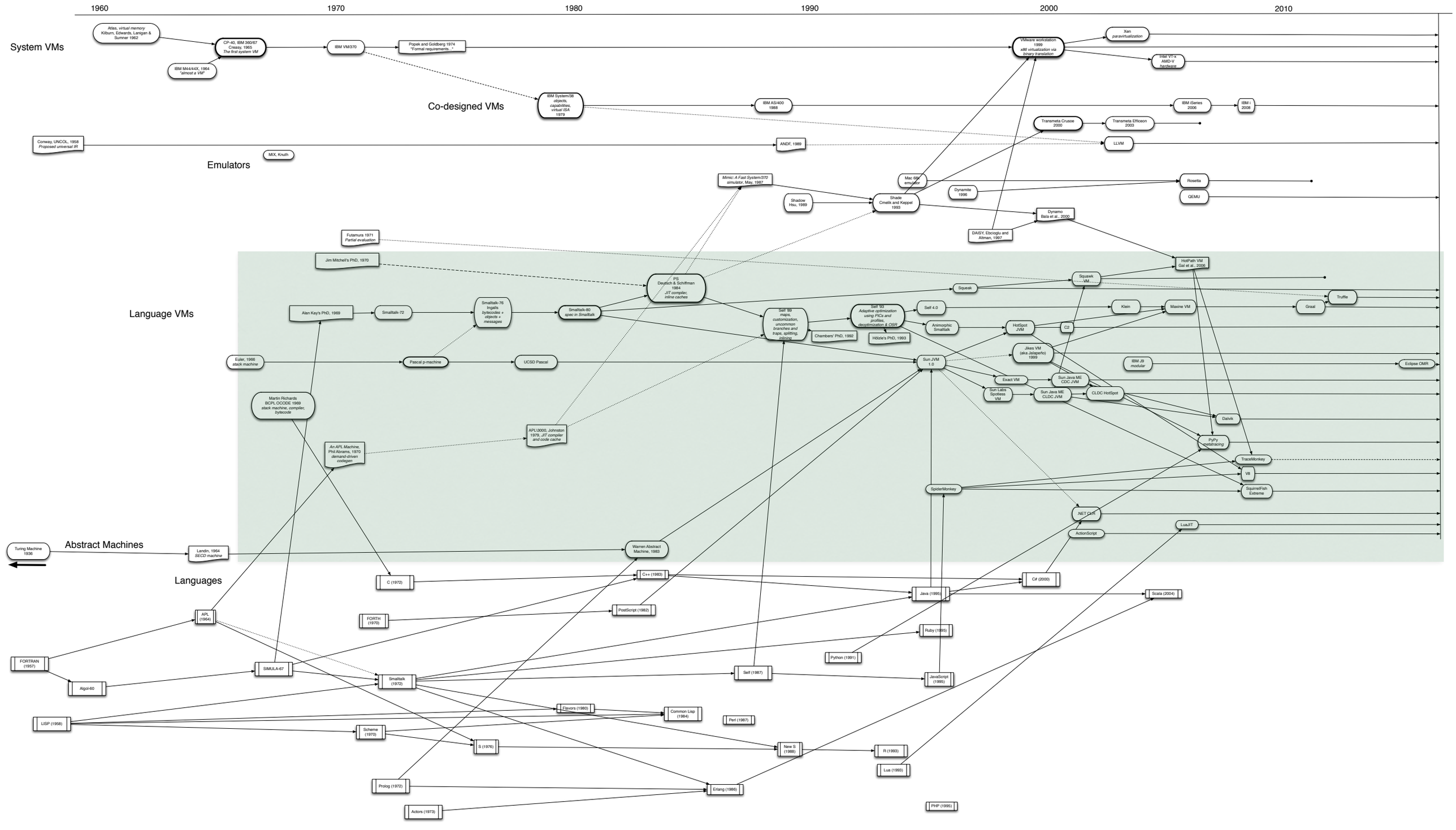
0.99 2016-05-20



omitted: GC, compilers, interpreters

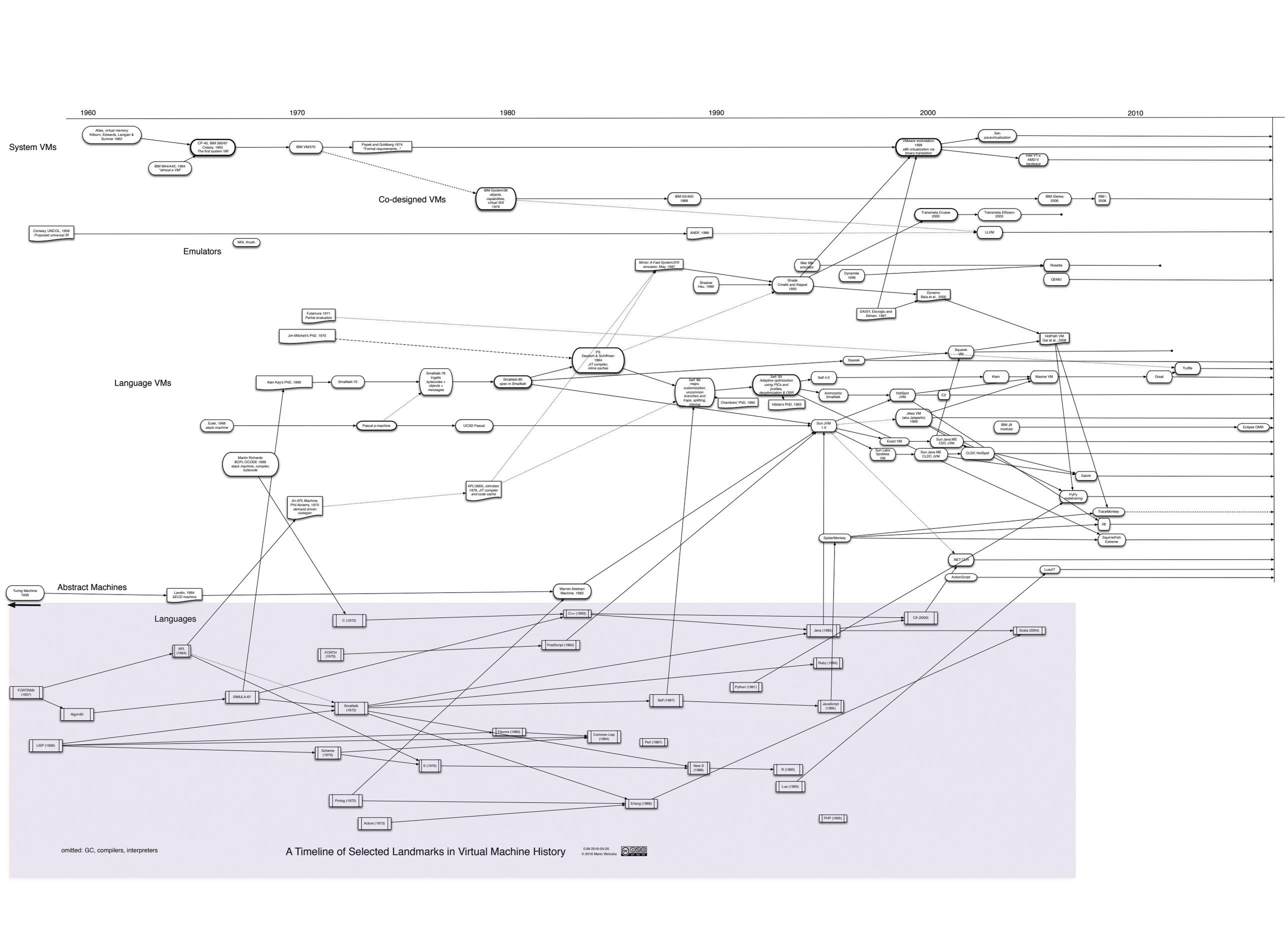
A Timeline of Selected Landmarks in Virtual Machine History

0.99 2016-05-20



omitted: GC, compilers, interpreters

A Timeline of Selected Landmarks in Virtual Machine History


1960
1970
1980
1990
2000
2010
System VMs
Co-designed VMs
Emulators
Language VMs
Abstract Machines
Languages
Atlas, virtual memory
Kilburn, Edwards, Lanigan & Sumner 1962
CP-40, IBM 360/67
Creasy, 1965
The first system VM
IBM M44/44X, 1964
"almost a VM"
IBM VM/370
Popek and Goldberg 1974
"Formal requirements..."
IBM System/38
objects, capabilities, virtual ISA
1979
IBM AS/400
1988
IBM iSeries
2006
IBM i
2008
VMware workstation
1999
x86 virtualization via binary translation
Xen
paravirtualization
Intel VT-x
AMD-V
hardware
Transmeta Crusoe
2000
Transmeta Efficeon
2003
Conway, UNCOL, 1958
Proposed universal IR
MIX, Knuth
ANDF, 1989
LLVM
Mimic: A Fast System/370 simulator, May, 1987
Shadow
Hsu, 1989
Shade
Cmelik and Keppel
1993
Mac 68K emulator
Dynamite
1996
Rosetta
QEMU
Dynamo
Bala et al., 2000
DAISY, Ebcioglu and Altman, 1997
Futamura 1971
Partial evaluation
Jim Mitchell's PhD, 1970
PS
Deutsch & Schiffman
1984
JIT compiler, inline caches
Squeak
Squawk VM
HotPath VM
Gal et al., 2006
Truffle
Graal
Alan Kay's PhD, 1969
Smalltalk-72
Smalltalk-76
Ingalls
bytecodes + objects + messages
Smalltalk-80
spec in Smalltalk
Self '89
maps, customization, uncommon branches and traps, splitting, inlining
Self '93
Adaptive optimization using PICs and profiles, deoptimization & OSR
Self 4.0
Chambers' PhD, 1992
Hölzle's PhD, 1993
Animorphic Smalltalk
HotSpot JVM
C2
Klein
Maxine VM
Euler, 1966
stack machine
Pascal p-machine
UCSD Pascal
Sun JVM 1.0
Jikes VM (aka Jalapeño)
1999
IBM J9
modular
Eclipse OMR
Exact VM
Sun Java ME CDC JVM
Sun Labs Spotless VM
Sun Java ME CLDC JVM
CLDC HotSpot
Dalvik
PyPy
meta-tracing
TraceMonkey
V8
SquirrelFish Extreme
SpiderMonkey
.NET CLR
ActionScript
LuaJIT
Martin Richards
BCPL OCODE 1969
stack machine, compiler, bytecode
An APL Machine, Phil Abrams, 1970
demand-driven codegen
APL\3000, Johnston 1979, JIT compiler and code cache
Turing Machine
1936
Landin, 1964
SECD machine
Warren Abstract Machine, 1983
C (1972)
C++ (1983)
C# (2000)
Java (1995)
Scala (2004)
APL (1964)
FORTH (1970)
PostScript (1982)
FORTRAN (1957)
Algol-60
SIMULA-67
Smalltalk (1972)
Self (1987)
Ruby (1995)
Python (1991)
JavaScript (1995)
LISP (1958)
Flavors (1980)
Common Lisp (1984)
Perl (1987)
Scheme (1970)
S (1976)
New S (1988)
R (1993)
Lua (1993)
Prolog (1972)
Erlang (1986)
Actors (1973)
PHP (1995)
omitted: GC, compilers, interpreters
A Timeline of Selected Landmarks in Virtual Machine History
0.99 2016-05-20
© 2016 Mario Wolczko

My path through the timeline

- I'll start by looking at early System VM history;
- Then: Language VMs,
- …before returning to recent System VMs,
- …and back to Language VMs,
- With excursions into other topics (Co-designed VMs, Emulators, etc.) en route.

System VMs

Part 1, 1964–1974

What is a System VM?

A System VM implements the *complete* hardware-software interface (user *and* privileged ISAs)

- Hence, can host an OS — like a real machine
- Must also emulate I/O and other device interfaces

The Early Days

- System VMs pre-date language VMs — how did they come about?

- Let's look at the early history…

The 1950s

- Computers were single-user devices.

- The OS was like a library, used to make programming easier (providing common routines, device abstractions, filing [tape, disk]).

- Some OSes also provided a batch job scheduling system.

The 1960s

- Early 1960s: time-sharing computer services were being adopted. There was a huge push to build time-sharing OSes (IBM System/360 being the prime example).

- Time-sharing was meant to optimize machine time, not user time: computation could be performed simultaneously with I/O. *Interactive* time-sharing came later.

- *Virtual memory* was invented (ATLAS, 1962) to make programming easier.

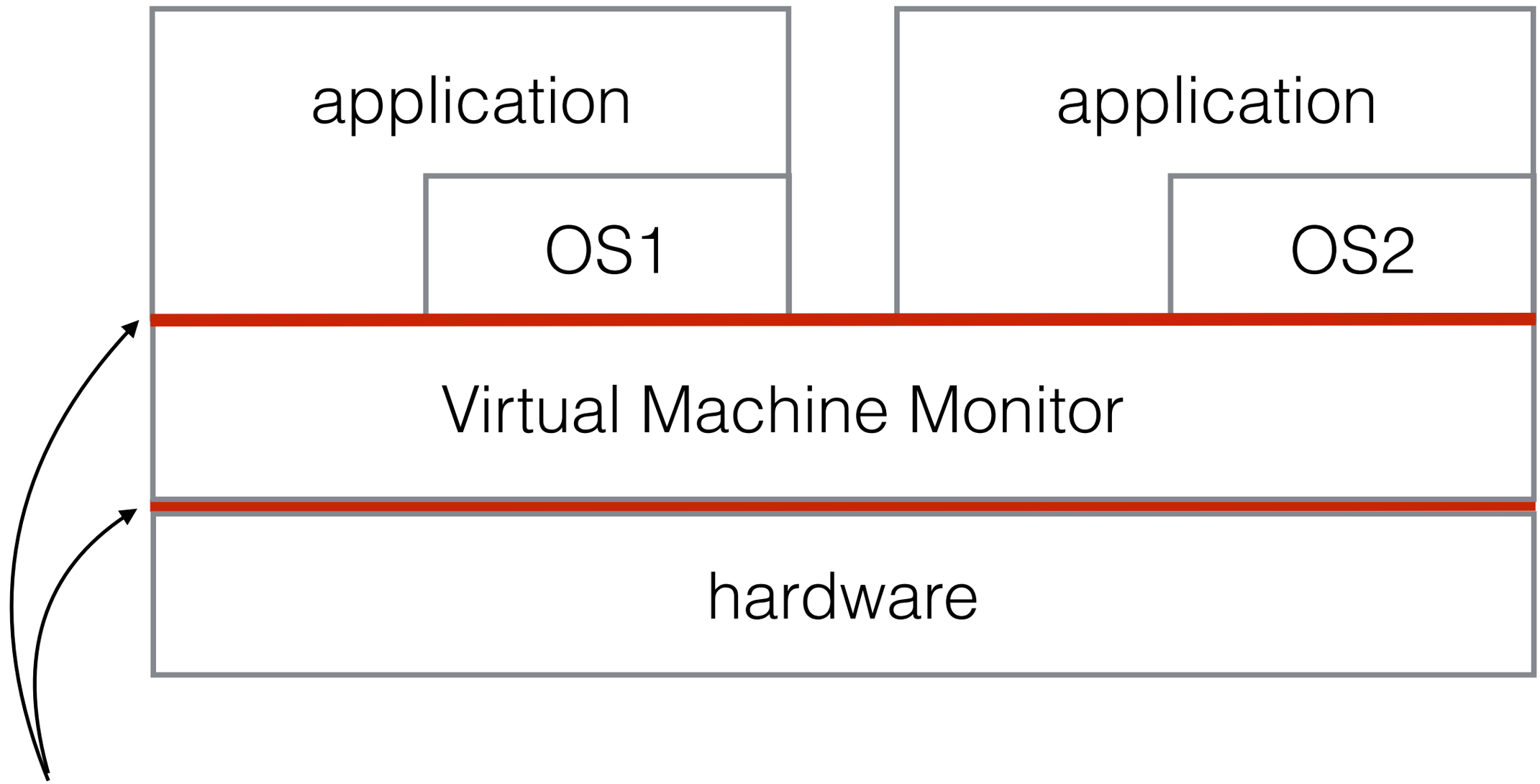
  - No need to manually swap code and data to/from disk/drum.

  - Originally called “one-level storage”

Time-sharing and Virtual Machines

- One approach to time-sharing is the one we are familiar with today: a single OS hosting multiple applications and users. But it was not the only approach.
- System VMs provided an alternative approach to time-sharing:
  - Every user/application had a dedicated, single-user OS.
  - Every user/application could have a *different* OS.

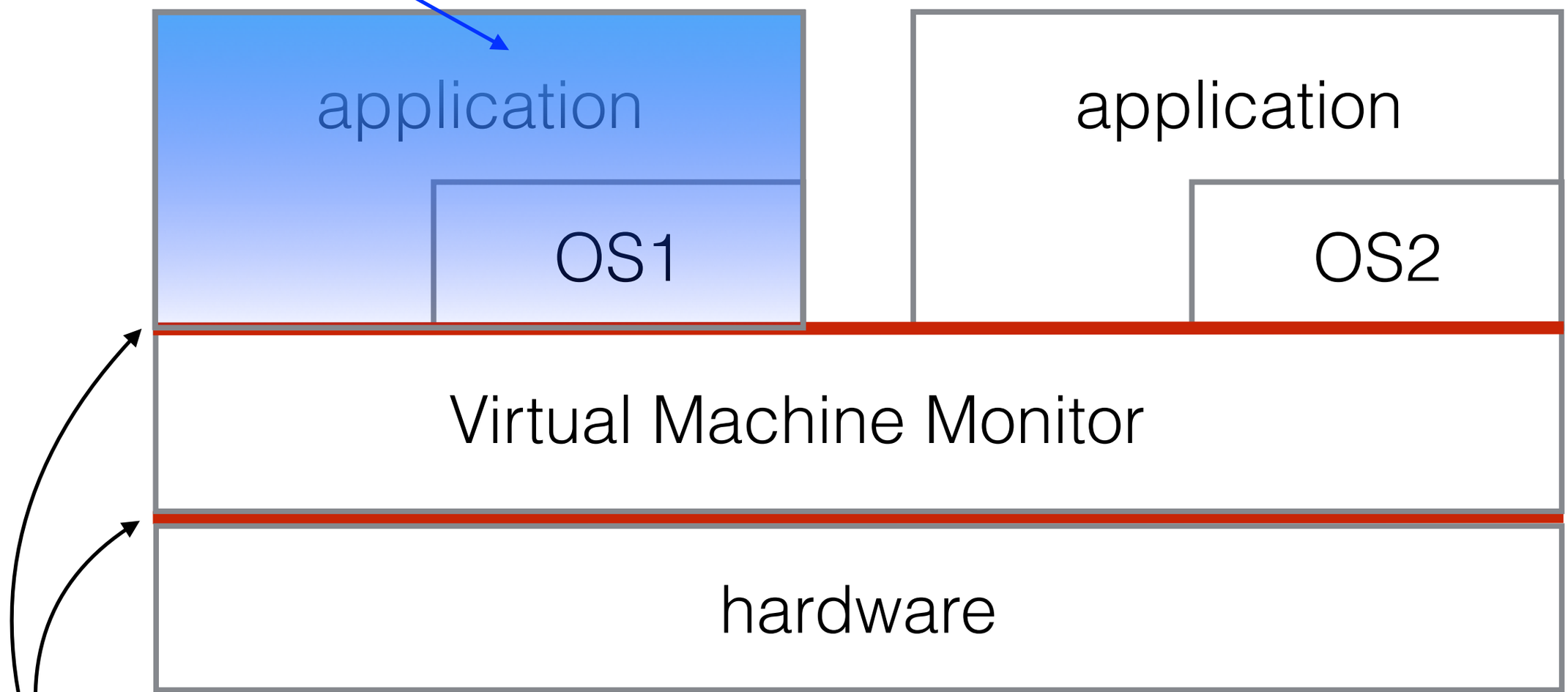
Early System VM architecture



Hardware and virtual ISAs are (almost) the same

Early System VM architecture

a “Virtual Machine”



Hardware and virtual ISAs are (almost) the same

The Virtual Machine Monitor

- A relatively small, “thin” layer with little performance impact
- Maintained the memory mappings
- Scheduled the VMs onto the CPU
- Enforced partitioning of physical resources (memory, devices)

Why use VMs for time-sharing?

- At first sight, this approach seems incredibly wasteful: each user/app has a copy of the OS, at a time when memory was extremely expensive.

- But: it solved an important problem: computers were extremely expensive and so down-time had to be avoided. Many installations had a single computer. How do you upgrade apps and OSes and remain in production?

CP-40 for the IBM System/360 Model 40

- Source: Melinda Varian, *VM and the VM Community: Past, Present and Future*, 1991, http://www.leeandmelindavarian.com/Melinda/neuvm.pdf, appendix by L.W. Comeau, *CP/40—The Origin of VM/370*

- CP-40 was the first implemented System VM (1965); an experimental predecessor (M44/44X) was “close”

- A system could host 14 VMs, each of 256KB virtual memory (128KB phys.mem. — 32 4KB pages)

- An experimental address translation mechanism was added to the hardware (without extending the cycle time!)

The user model

- Each VM ran a single user OS, CMS. Access to devices was mediated by job control.

- Paravirtualization provided VM services (more later)

- Originally built as a measurement platform, to measure the behavior of (non-VM) programs in a VM environment. Goal was to determine best page size, time slice, etc.

- MIT-centric early history

The 1970s

- By 1974, the basic ideas of system VMs were well understood, and formal virtualizability requirements were described by Popek and Goldberg in a CACM paper, *Formal Requirements for Virtualizable Third Generation Architectures*.
    - Basic idea: instructions should compose with a VM correctly, or trap so that the VM can “do the right thing”. Many ISAs of the time did not meet this requirement.
- System/370 however, did, and System VMs were commonplace. Similar ideas were adopted in other mainframe OSes. These systems had legendary reliability.

1980s: The quiet period

- VMMs used in mainframes, for decades. No fuss. No academic interest.
- The rest of the computing world adopted the single OS model, quietly ignoring or forgetting about System VMs.
- More to follow…meanwhile, let's talk a look at early language VMs.

Language VMs

Part 1, 1966—circa 2000

What is a Language VM?

- A language-specific Process VM
    - The VM presents an OS-like interface to applications as well as an ISA
- Often created together with, or during the evolution, of the associated language.
- Typically embodies language-specific concepts and semantics.
    - A relatively small jump from language semantics to VM interface.

Timeline

- BCPL
- Pascal
- Smalltalk
- Self
- Java
- JavaScript

BCPL

- Basic CPL (Combined Programming Language)
  - CPL was a broad-spectrum language conceived by Christopher Strachey at Cambridge and others in the early 1960s.
  - Martin Richards* (Cambridge) designed the BCPL subset [Basic CPL] in 1966 (which was implemented in 1967)
- Used for systems programming (compilers, operating systems)
- BCPL was a major influence on the design of C
- The compiler emitted OCODE, which could be translated to native machine code; Cintcode was a bytecode for interpretation

* Of Richards benchmark fame

OCODE

From *The BCPL Cintsys and Cintpos User Guide* — http://www.cl.cam.ac.uk/users/mr10/

- Simple! 10 page definition.
- Not very language-oriented.
- Main aim was easy porting of the compiler — which was achieved.

8.2 The OCODE Abstract Machine

OCODE was specifically designed for BCPL and is a compromise between the desire for simplicity and the conflicting demands of efficiency and machine independence. OCODE is an assembly language for an abstract stack based machine that has a global vector and an area of memory for program and static data as shown in figure 8.2.

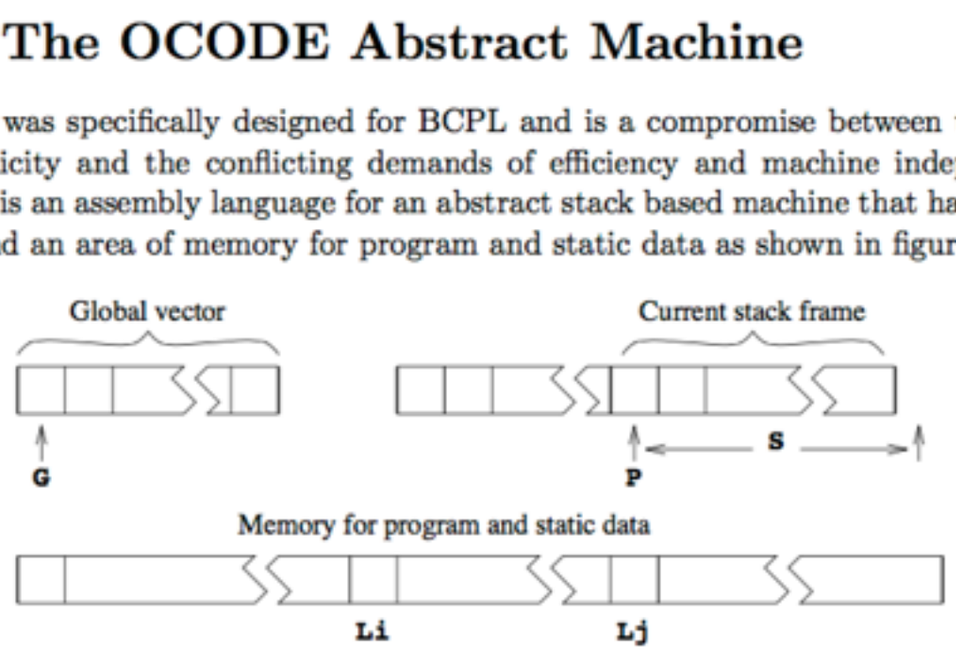
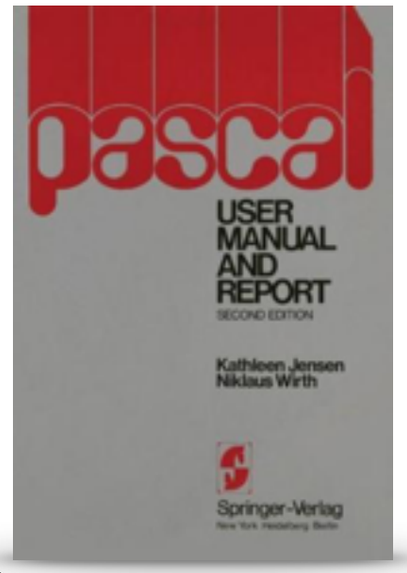


Figure 8.2: The BCPL abstract machine

The global vector is pointed to by the G pointer and the current stack frame is pointed to by the P pointer. S is the size of the current stack frame, and so P!S is the first free element of the stack. The value of S is always known during compilation and so is not held in a register of the OCODE abstract machine machine. Any assignments

Pascal p-code



- Origins in a Pascal compiler developed in the mid-1970s at ETH Zurich
- Used in the UCSD p-System (OS) released in 1978, deployed widely for commercial use
- Stack machine, very simple, originally interpreted
- Later: Hardware implementations: Western Digital's Pascal MicroEngine, NCR, later Lilith (Modula-2 M-code)

References

- Urs Ammann, *On Code Generation in a PASCAL Compiler*, Software: Practice and Experience, 7(3) 1977, pp.391—423

- The UCSD P-System Museum, http://www.threedee.com/jcm/psystem/

- Wikipedia entries for:
  - *UCSD Pascal*
  - *Joel McCormack*, designer of the NCR p-machine. Includes overview of architecture and microcode.
  - *Pascal MicroEngine* — microcoded interpreter
  - *P-code machine* — includes source of Wirth's simple p-machine
  - *Business Operating System* — a VM/OS for COBOL
  - *Lilith*

Abstract machines

- Perhaps a better name than virtual machine?

abstract: adj. existing in thought or as an idea not having a physical or concrete existence

Examples:

- Landin's SECD machine for Lambda Calculus
- ..and of course, the Turing machine

Details, details

- However, some details, while irrelevant to the semantics of the guest language, are pragmatically essential
    - Examples: instruction encodings, for binary distribution and inter-operability
- At the other extreme, the x86 ISA is an “abstract machine” — a Xeon, e.g., is one concrete embodiment.
- The problem with the Turing machine and the SECD machine is that they are *too* abstract. To make a useful execution engine, some things cannot be abstract, but must be concrete.
- A “concrete abstract machine”? The term *virtual machine* has stuck.

  virtual: adj. not physically existing as such but made by software to appear to do so

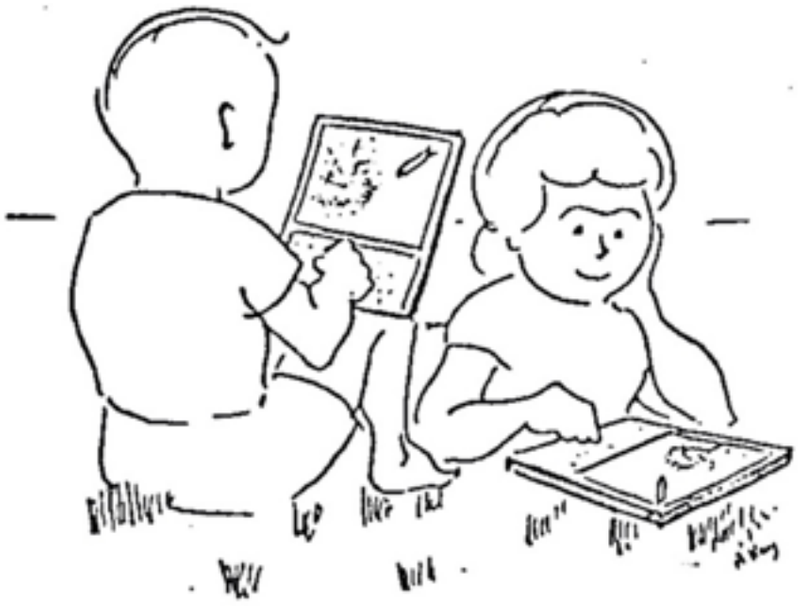
  cf. *virtual image* (optics)

The Warren Abstract Machine for Prolog

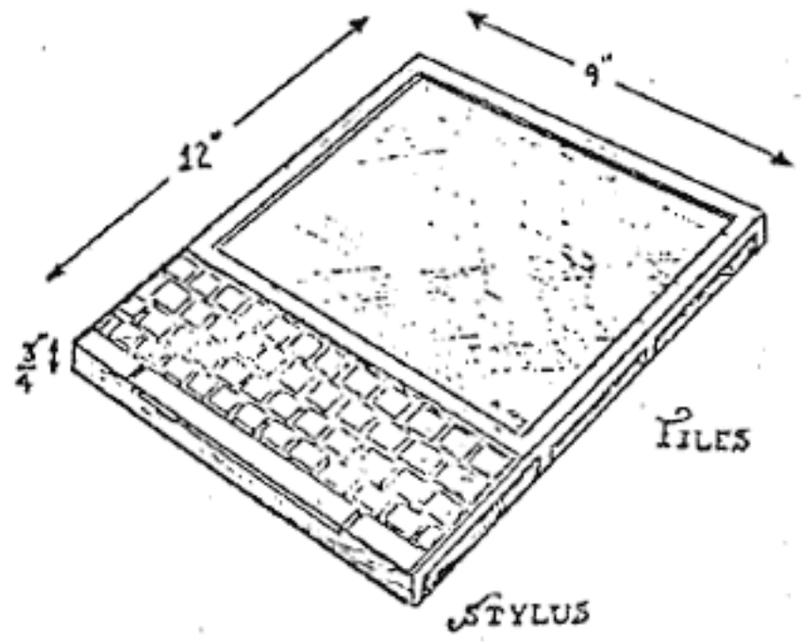
- David Warren, SRI technical note 309, *An Abstract Prolog Instruction Set*, 1983
- In addition to a heap and a stack there is a *trail* and a *Push-Down List* (PDL).
- The stack contains environments and *choice points*.
- The trail keeps track of which bindings have to be retracted after a clause fails
- The PDL contains pairs of nodes which have to be considered for unification.
- I would consider this to be a language VM, or very close.

Smalltalk

- From the mid-1970s to the mid-1980s, Smalltalk took up the running in VM technology.



It’s 1969…

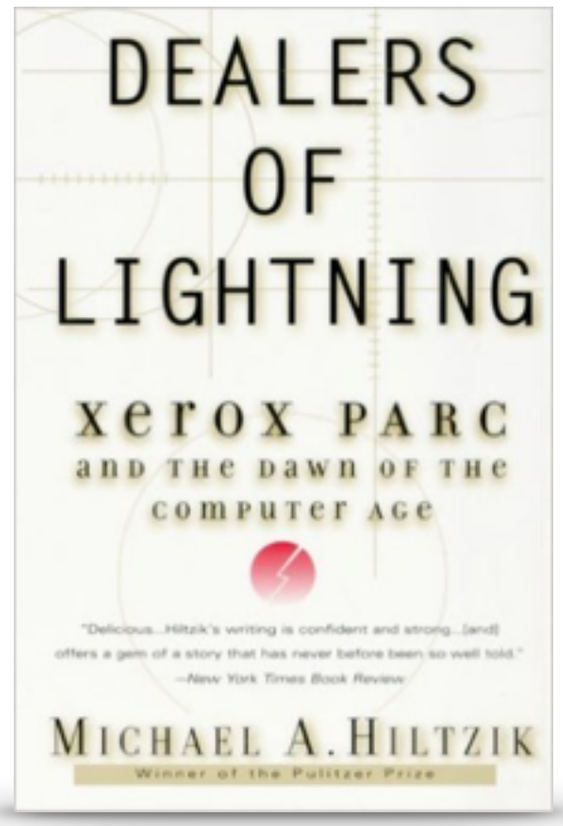


- Alan Kay’s Ph.D. thesis, *The Reactive Engine*, describes a future of personal, portable computers and speculates on how they will be programmed and used.

Early 1970s

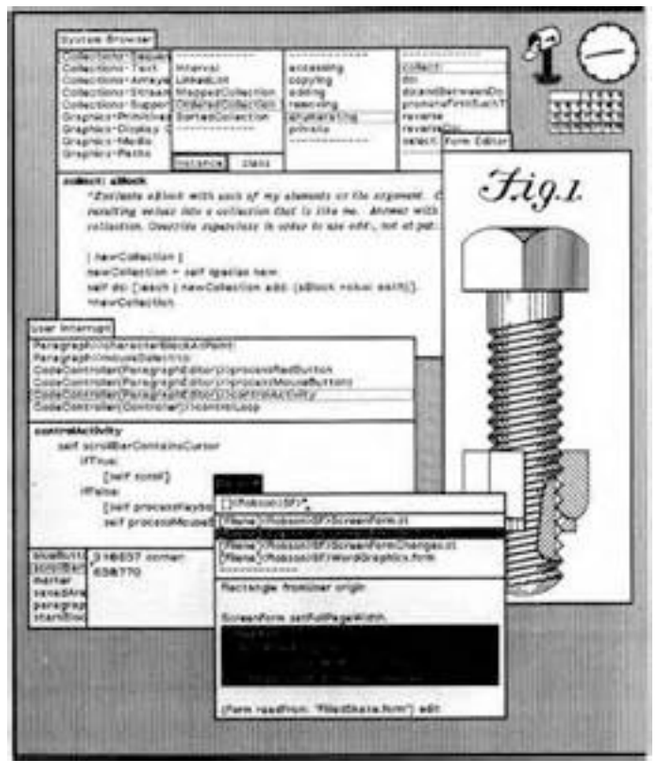
- In 1970, Kay joins Xerox PARC (just created). Forms the Learning Research Group, attracts other researchers, including Adele Goldberg.

- The Smalltalk language and system are invented and developed through several versions. The aim is to build a system capable of being used by children to learn.



Smalltalk-76 and the Alto

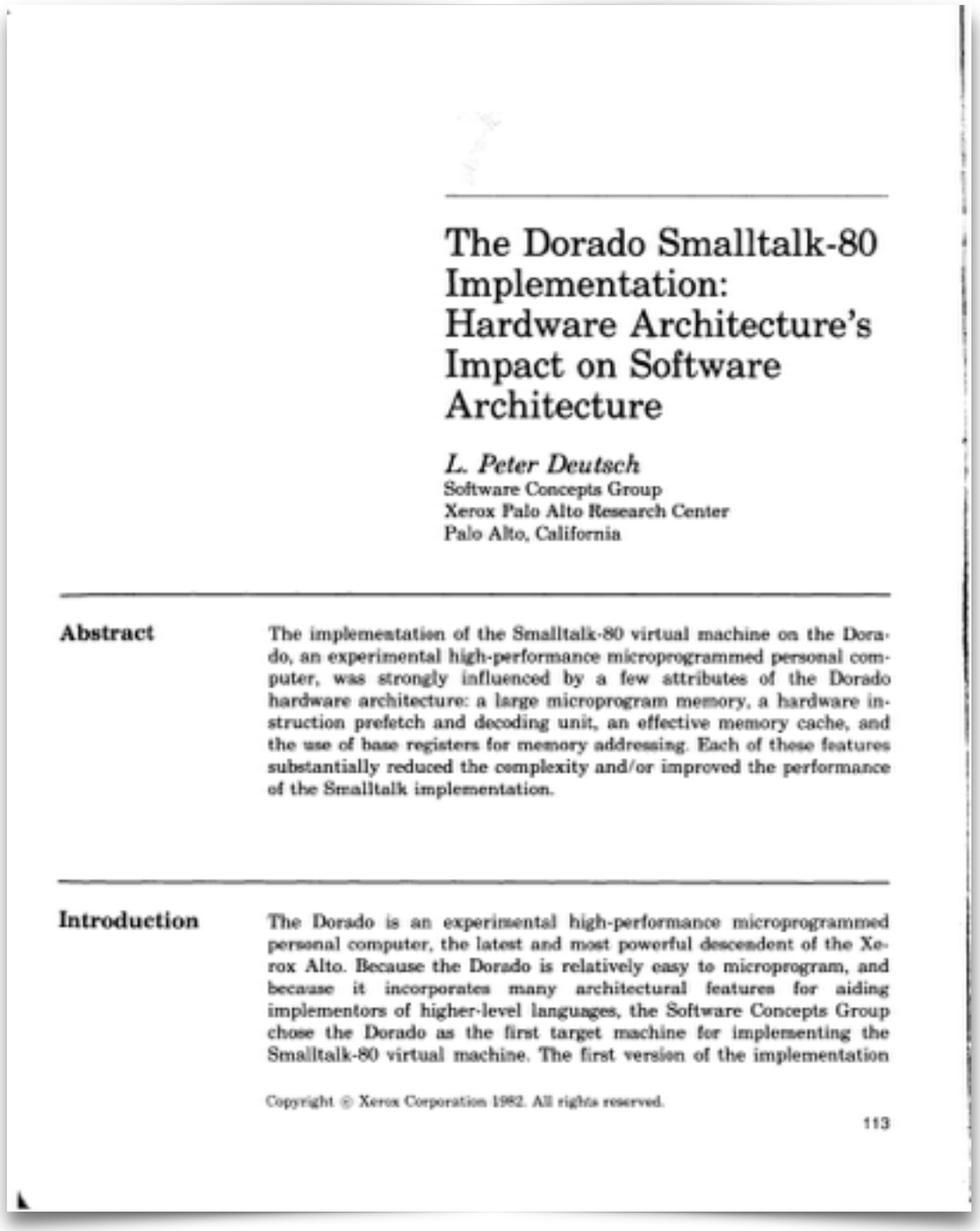
- PARC develops the Alto workstation — the “interim Dynabook” — a personal computer with high-resolution bitmapped graphics, local storage and a fast network connection.
- Smalltalk-76 is honed for the Alto. BitBlt, copy and paste are invented.



Smalltalk-80 and the Dorado

- The Dorado follows the Alto: bigger, faster.

- The Smalltalk-80 VM is ported to the Dorado (microcoded bytecodes)



The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture

L. Peter Deutsch
Software Concepts Group
Xerox Palo Alto Research Center
Palo Alto, California

Abstract

The implementation of the Smalltalk-80 virtual machine on the Dorado, an experimental high-performance microprogrammed personal computer, was strongly influenced by a few sttributes of the Dorado hardware architecture: a large microprogram memory, a hardware instruction prefetch and decoding unit, an effective memory cache, and the use of base registers for memory addressing. Each of these features substantially reduced the complexity and/or improved the performance of the Smalltalk implementation.

Introduction

The Dorado is an experimental high-performance microprogrammed personal computer, the latest and most powerful descendent of the Xerox Alto. Because the Dorado is relatively easy to microprogram, and because it incorporates many architectural features for aiding implementors of higher-level languages, the Software Concepts Group chose the Dorado as the first target machine for implementing the Smalltalk-80 virtual machine. The first version of the implementation

1981–3: Smalltalk-80 is released to the world

- *BYTE* special issue (Aug 1981)

Building Control Structures in the Smalltalk-80 System

L Peter Deutsch
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Just as *data structures* refer to the ways that we group data together by using simple *objects* to represent more complex objects, *control structures* refer to the ways a programmer can build up complex sequences of *operations* from simpler ones. The easiest example of a control structure is sequencing: do something and then do something else. Two other familiar examples are the *conditional* structure (if some condition is true, do

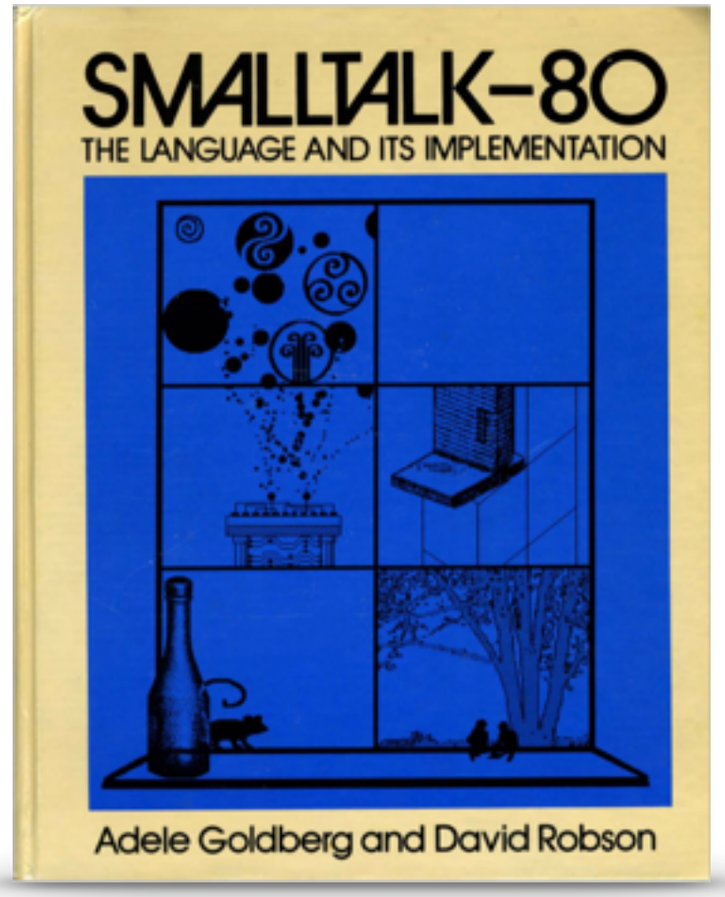
and the simple loop:

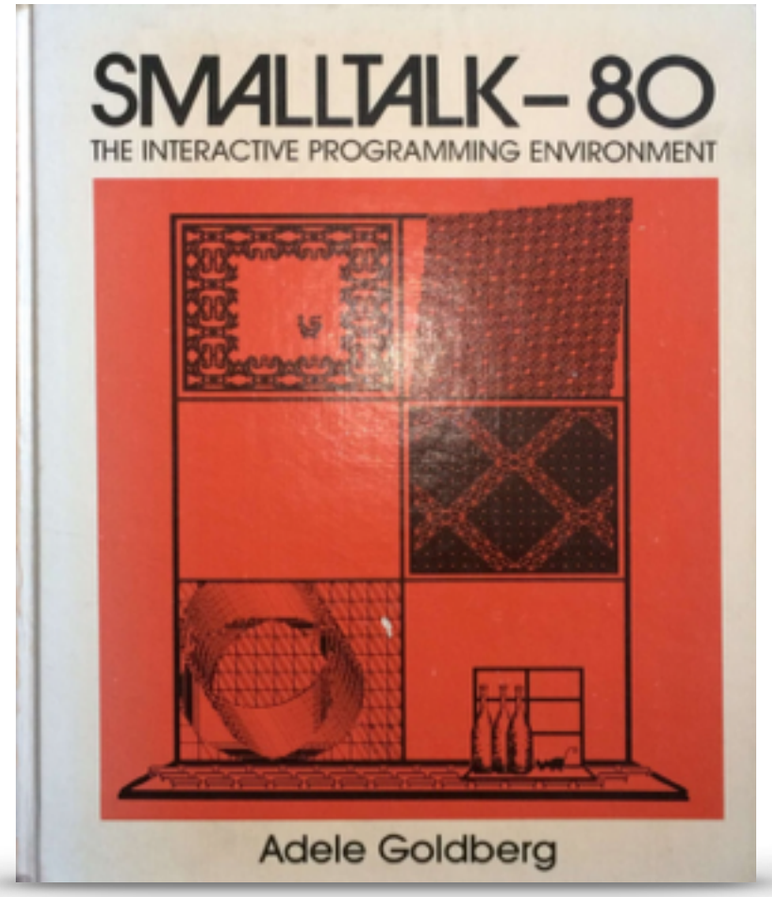
```
[someCondition] whileTrue: [somethingToDo]
[someCondition] whileFalse: [somethingToDo]
```

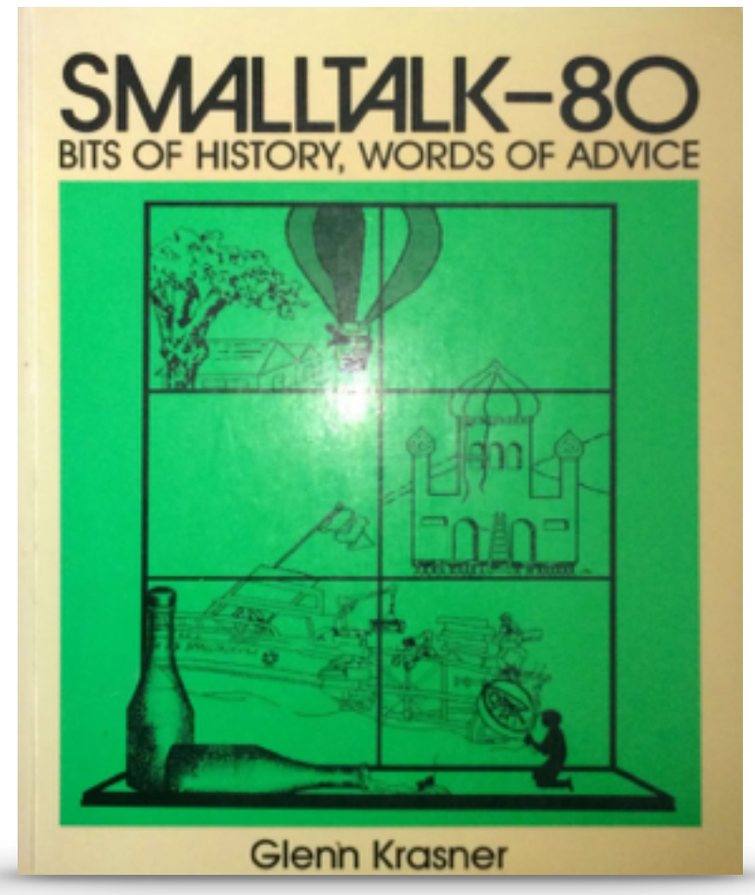
The most powerful tool for building new structures is the *block*. Two examples are:

1981–3: Smalltalk-80 is released to the world

- The “colored books” (1983)





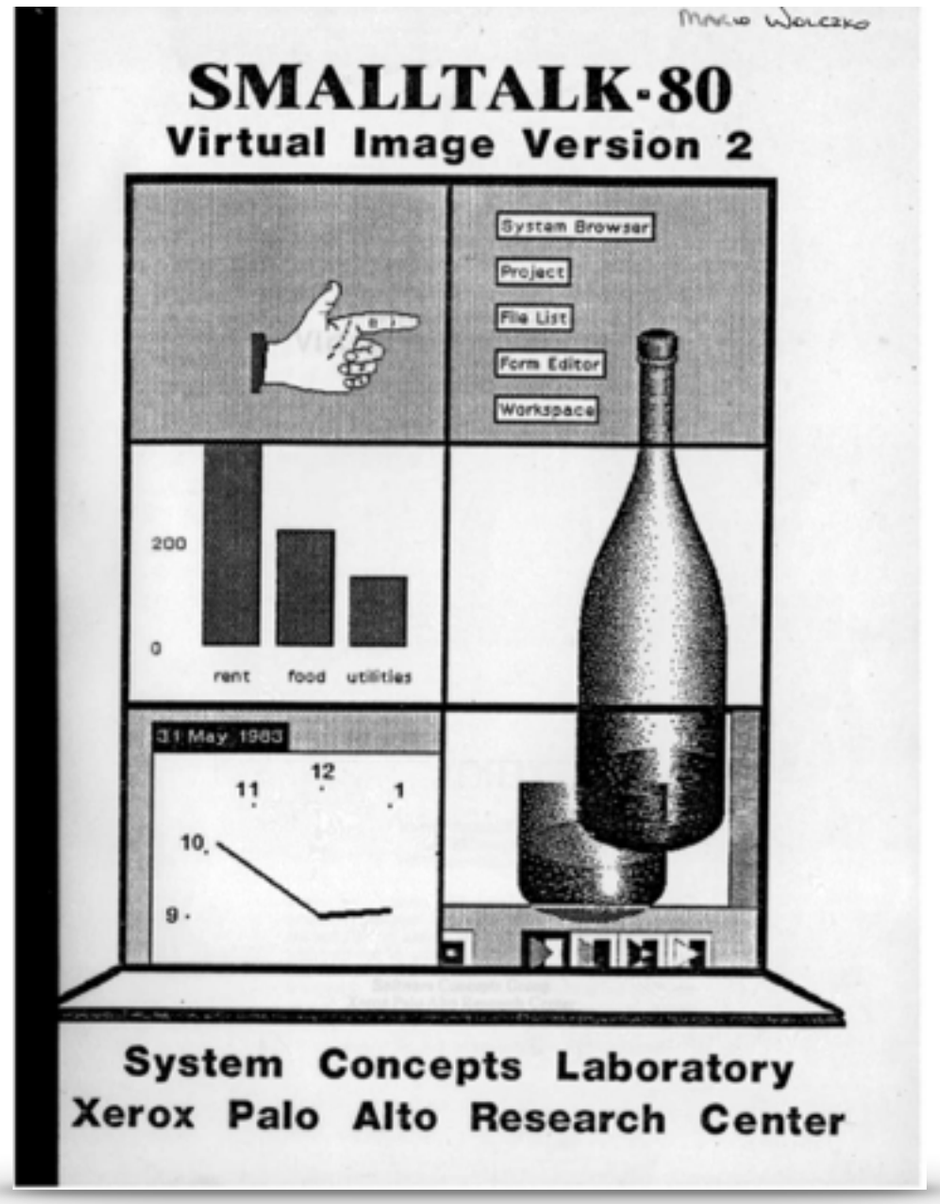


Smalltalk-80 Virtual Machine

- Defined by a reference implementation (in Smalltalk!) in the Blue Book
- Bytecode ISA, object memory
    - Classes, metaclasses, method activations are objects!

1981–3: Smalltalk-80 is released to the world

- The tape (1983)
- All the objects
- Roll your own VM!
  - Slow! (see Green Book)



PS

- *Efficient Implementation of the Smalltalk-80 System*, L Peter Deutsch and Allan M Schiffman, POPL 1984

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and

machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure

The paper: contributions

- Just-In-Time translation of Smalltalk bytecode to machine code; code caching and lookup
- Inline caching of message send targets
- On-demand conversion of contexts (activation records) from on-stack to hybrid and heap-allocated forms
- Implemented deferred reference counting (described in a 1976 paper by Deutsch & Bobrow)
- For more detail, watch CS294 session on youtube.

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call *Lookup*
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver*
(generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver*
(generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

nmethod n_1 *for* f *in class of* p, ***Q****:*
entry point:
  if receiver-class != ***Q*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_0
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

nmethod n_1 *for* f *in class of* p, ***Q****:*
entry point:
  if receiver-class != ***Q*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_1
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

nmethod n_1 *for* f *in class of* p, ***Q****:*
entry point:
  if receiver-class != ***Q*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_1
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

nmethod n_1 *for* f *in class of* p, ***Q****:*
entry point:
  if receiver-class != ***Q*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Inline caches (simplified)

…
call site for p.f() *(*p f *in Smalltalk/Self)*:
; code to put *p* in *receiver* register
; code to put *'f'* in method *name* register
call n_1
…

Lookup:
Find nmethod *n* associated with *name* in *receiver* (generate code if necessary).
Patch call site to link to entry point of *n.*
Jump to verified entry point of *n*.

nmethod n_0 *for* f *in class of* p, ***P****:*
entry point:
  if receiver-class != ***P*** jump *Lookup*
verified entry point:
  …rest of nmethod…

nmethod n_1 *for* f *in class of* p, ***Q****:*
entry point:
  if receiver-class != ***Q*** jump *Lookup*
verified entry point:
  …rest of nmethod…

Influences

- Mitchell's 1971 Ph.D. thesis
- BCPL/Pascal P-code/Forth/Lisp
- Dynamic code generation:
  - Rob Pike's BitBLT for the Blit terminal (1982)
  - Regexp compilation
- APL compilers? Not really.

Self

1987–1995

- Language designed in 1987 as a successor to Smalltalk; even simpler and more regular

- Objects, slots, methods, messages

- VM had only 8 bytecodes!

- Stanford & PARC 1987—1992
  Sun Labs 1992—1995

Self implementation innovations

From JIT to adaptive, feedback-driven optimization (to come after JIT compilation):

- PICs, maps, generational heap
- C++ implementation tricks
- Optimizing compilation of a dynamic language
- Type feedback
- Adaptive optimization

Self implementations 1989—1992

VM structure
Elgin Lee's thesis, 1988

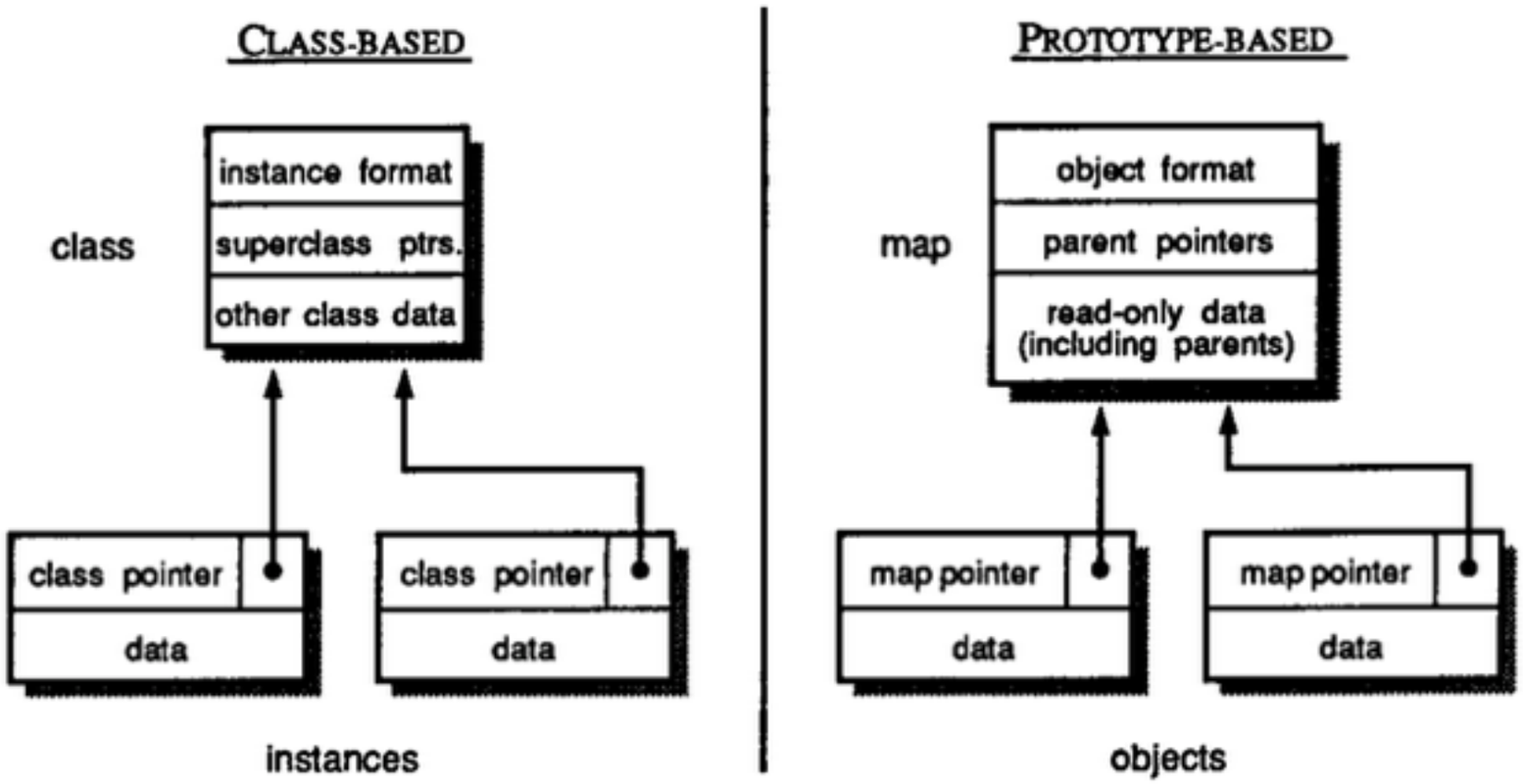
- Generational heap
- Maps
- C++ representation
- Code dependencies

Compiler optimizations
Craig Chambers' thesis, 1992

- Type analysis
- Customization
- Splitting
- Together with inlining, enabled big performance gains

Maps

- Instead, we factor out the shareable part into a *map.* In later VMs this is called a *hidden class.*



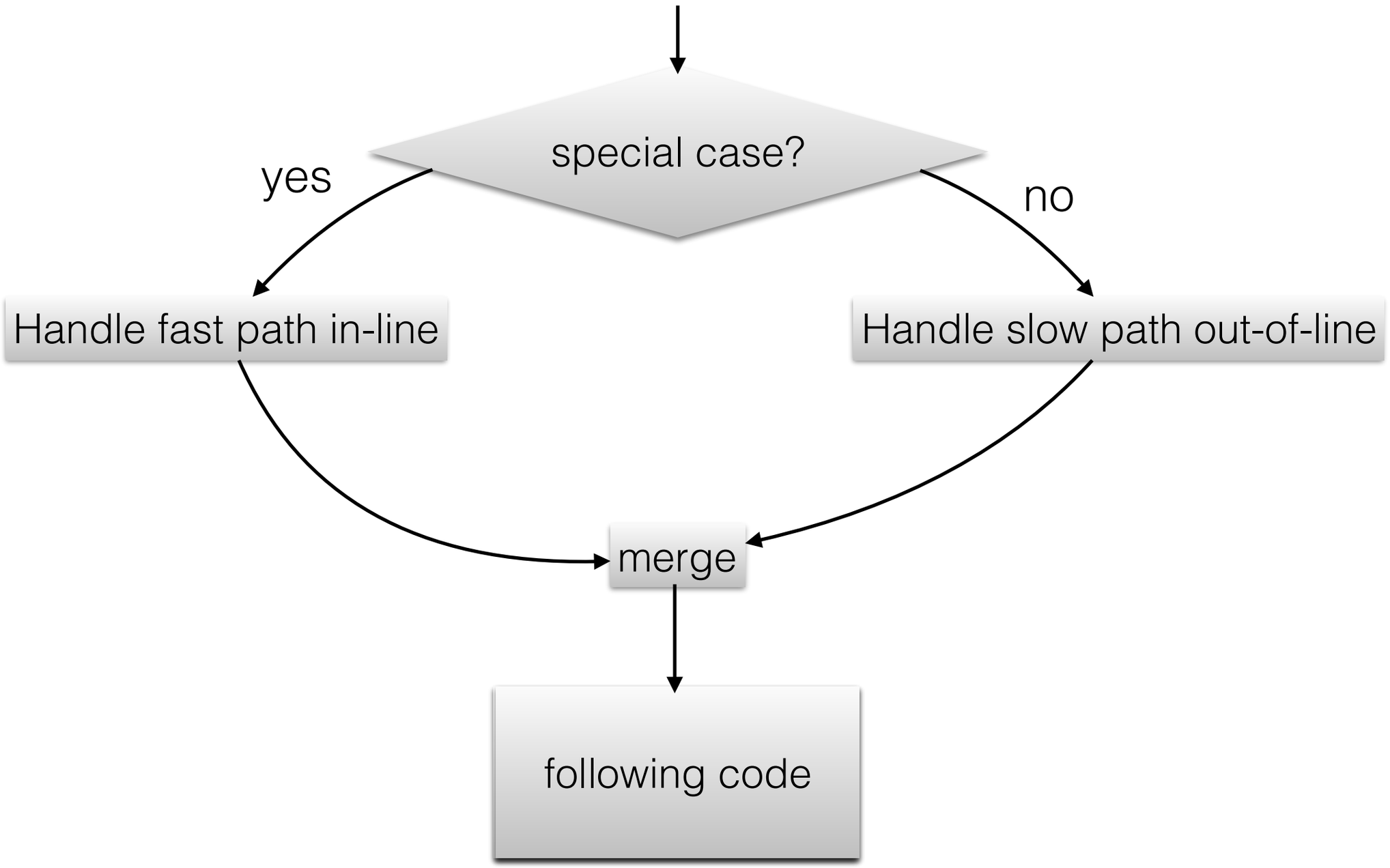
Map internals

- Each map (`objects/map.hh`) contains a list of slot descriptors (`objects/slotDesc.hh`), each of which names and categorizes the corresponding slot (`objects/slotType.hh`):
  - *assignable* data slot (has corresponding word in object as indicated offset), or *constant* data slot (value is in slotDesc), or *argument* slot (methods only)
  - is it a parent slot?

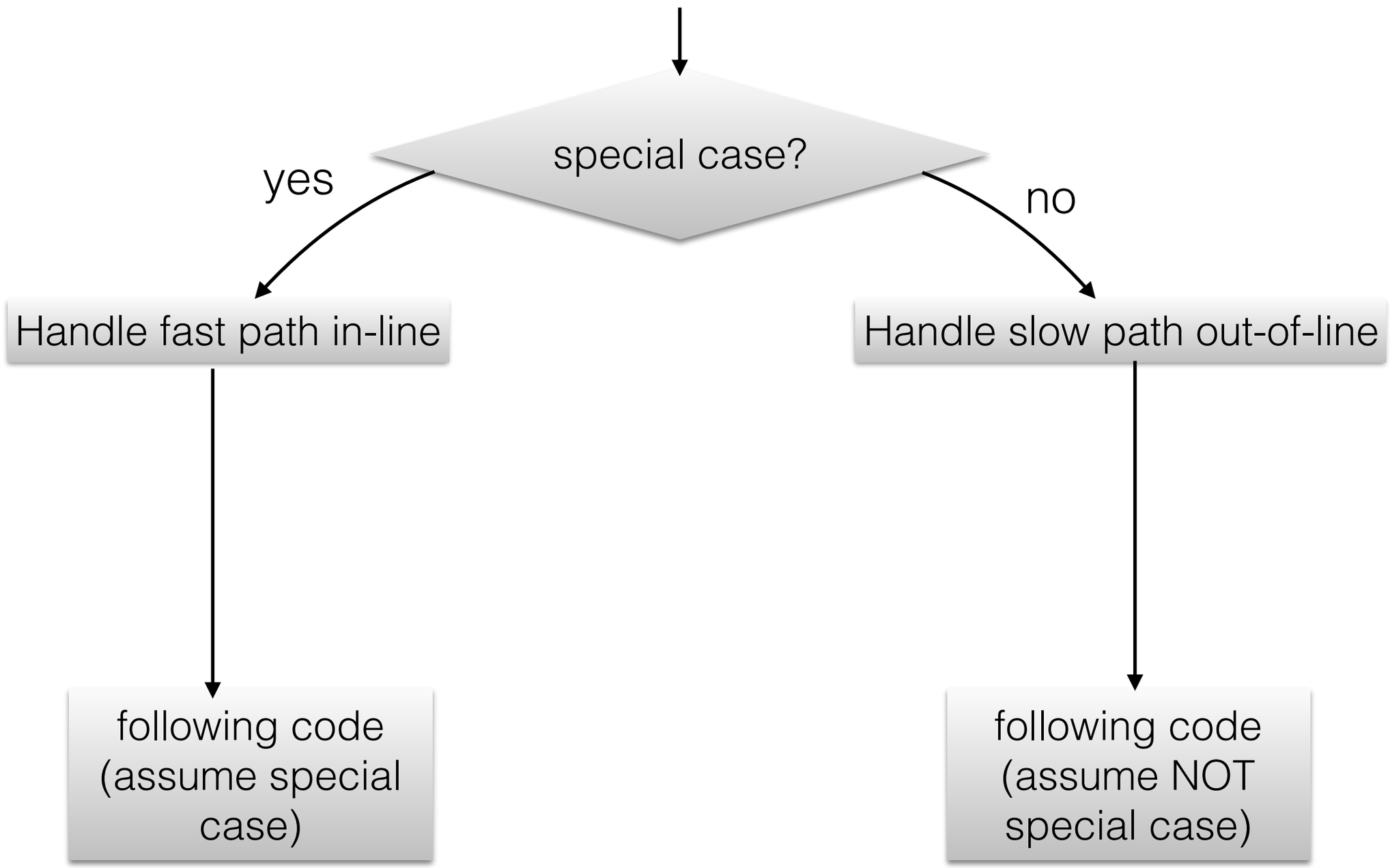
The map table

- When an object is cloned, it shares its map with its clone.
- When objects are altered using the *programming primitives* (which can, e.g., add a new slot to an object), a new map is created, but checked against a canonical map table (`memory/mapTable.hh`) to ensure that all maps are structurally unique.

Splitting


special case?
yes
no
Handle fast path in-line
Handle slow path out-of-line
merge
following code

Splitting


special case?
yes
no
Handle fast path in-line
Handle slow path out-of-line
following code (assume special case)
following code (assume NOT special case)

Example: *sumTo:* in Self

1 sumTo: 5 ⇒ 1+2+3+4+5

sumTo: calls *to:Do:* calls *to:By:Do* whose inner loop is:

```
[i <= end] whileTrue: [
    block value: i.
    i: i + step]
```

and bytecode is:

```
pushLiteral: [i <= end]
pushLiteral: [block value: i. …]
send whileTrue:
```

i.e., there is no explicit control structure at all!

```
1 sumTo: 5
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   to: upperBound Do: [| :index |
      sum: sum + index].
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   to: upperBound Do: [| :index |
      sum: sum + index].
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   to: upperBound Do: [| :index |
      sum: sum + index].
   sum)
```

```
to: end Do: block = (
  to: end By: 1 Do: block
)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   to: upperBound By: 1 Do: [| :index |
      sum: sum + index].
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   to: upperBound By: 1 Do: [| :index |
      sum: sum + index].
   sum)
```

```
to: end By: step Do: block = (
  step = 0 ifTrue: [error: 'step is zero']
    False: [
      step < 0 ifTrue: […step down…]
         False: [| i |
            i: self.
            [ i<=end ] whileTrue: [
            block value: i
            i: i + step ] ] ]
)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
  1 = 0 ifTrue: [error: 'step is zero']
    False: [
      1 < 0 ifTrue: […step down…]
        False: [| i |
          i: self.
          [ i<=upperBound ] whileTrue: [
            [| :index | sum: sum + index] value: i.
            i: i + 1 ] ] ]
  sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
  1 = 0 ifTrue: [error: 'step is zero']
    False: [
      1 < 0 ifTrue: […step down…]
        False: [| i |
          i: self.
          [ i<=upperBound ] whileTrue: [
            [| :index | sum: sum + index] value: i.
            i: i + 1 ] ] ]
  sum)
```

```
= aNumber = (
  _IntegerEQPrimitive: aNumber
    ifFail: […])
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
  false ifTrue: [error: 'step is zero']
    False: [
      1 < 0 ifTrue: […step down…]
        False: [| i |
          i: self.
          [ i<=upperBound ] whileTrue: [
            [| :index | sum: sum + index] value: i.
            i: i + 1 ] ] ]
  sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
  false ifTrue: [error: 'step is zero']
    False: [
      1 < 0 ifTrue: […step down…]
        False: [| i |
          i: self.
          [ i<=upperBound ] whileTrue: [
            [| :index | sum: sum + index] value: i.
            i: i + 1 ] ] ]
  sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
  false ifTrue: [error: 'step is zero']
    False: [
      1 < 0 ifTrue: […step down…]
        False: [| i |
          i: self.
          [ i<=upperBound ] whileTrue: [
            [| :index | sum: sum + index] value: i.
            i: i + 1 ] ] ]
  sum)
```

```
ifTrue: trueBlock False: falseBlock = (
  falseBlock value
)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   [1 < 0 ifTrue: […step down…]
         False: [| i |
            i: self.
            [ i<=upperBound ] whileTrue: [
               [| :index | sum: sum + index] value: i.
               i: i + 1 ] ] ] value
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   [1 < 0 ifTrue: […step down…]
         False: [| i |
            i: self.
            [ i<=upperBound ] whileTrue: [
               [| :index | sum: sum + index] value: i.
               i: i + 1 ] ] ] value
   sum)
```

```
value = (  _Value )
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   1 < 0 ifTrue: […step down…]
         False: [| i |
            i: self.
            [ i<=upperBound ] whileTrue: [
               [| :index | sum: sum + index] value: i.
               i: i + 1 ] ]
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
  [ i<=upperBound ] whileTrue: [
    [| :index | sum: sum + index] value: i.
    i: i + 1 ] ]
  sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
  [ i<=upperBound ] whileTrue: [
    [| :index | sum: sum + index] value: i.
    i: i + 1 ] ]
  sum)
```

```
whileTrue: block = (
  [ value ifTrue: block False: [^nil]
  ] loop
)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [[ i<=upperBound ] value ifTrue: [
         [| :index | sum: sum + index] value: i.
          i: i + 1 ]
      False: [^nil].
    ] loop.
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [[ i<=upperBound ] value ifTrue: [
         [| :index | sum: sum + index] value: i.
          i: i + 1 ]
      False: goto exit.
    ] loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [[ i<=upperBound ] value ifTrue: [
         [l :index l sum: sum + index] value: i.
          i: i + 1 ]
      False: goto exit.
    ] loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [i<=upperBound ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    ] loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [i<=upperBound ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    ] loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   [i<=upperBound ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    ] loop.
exit:
   sum)
```

```
loop = (  value. _Restart )
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   i <= upperBound ifTrue: [
          sum: sum + i.
           i: i + 1 ]
       False: goto exit.
    goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   i <= upperBound ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
       False: goto exit.
    goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
       False: goto exit.
    goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       temp := int(i) <= int(upperBound)
     else
       temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    goto loop.
exit:
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       temp := int(i) <= int(upperBound)
     else
       temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    goto loop.
exit:
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         temp := true
       else
         temp := false
     else
       temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    goto loop.
exit:
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         temp := true
       else
         temp := false
     else
       temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    goto loop.
exit:
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         temp := true
       else
         temp := false
     else
       temp := int(i) <= upperBound
   else
     temp := i <= upperBound
   temp ifTrue: [
         sum: sum + i.
          i: i + 1 ]
      False: goto exit.
    goto loop.
exit:
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum: sum + i.
         i: i + 1
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum: sum + i.
         i: i + 1
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum := int(sum) + int(i).
         if overflow(sum) then uncommon-trap
         i: i + 1
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum := int(sum) + int(i).
         if overflow(sum) then uncommon-trap
         i: i + 1
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum := int(sum) + int(i).
         if overflow(sum) then uncommon-trap
         i := int(i) + 1
         if overflow(i) then uncommon-trap
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(i)
     if hasIntegerTag(upperBound)
       if  int(i) <= int(upperBound)
         sum := int(sum) + int(i).
         if overflow(sum) then uncommon-trap
         i := int(i) + 1
         if overflow(i) then uncommon-trap
       else
         goto exit
     else
       uncommon-trap
   else
     uncommon-trap
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(upperBound)
     if  int(i) <= int(upperBound)
     sum := int(sum) + int(i).
     if overflow(sum) then uncommon-trap
     i := int(i) + 1
     if overflow(i) then uncommon-trap
     else
       goto exit
   else
       uncommon-trap
  goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
loop:
   if hasIntegerTag(upperBound)
     if  int(i) <= int(upperBound)
     sum := int(sum) + int(i).
     if overflow(sum) then uncommon-trap
     i := int(i) + 1
     if overflow(i) then uncommon-trap
     else
       goto exit
   else
       uncommon-trap
  goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   if !hasIntegerTag(upperBound) uncommon-trap
loop:
   if  int(i) <= int(upperBound)
     sum := int(sum) + int(i).
     if overflow(sum) then uncommon-trap
     i := int(i) + 1
     if overflow(i) then uncommon-trap
   else
     goto exit
   goto loop.
exit:
   sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   if !hasIntegerTag(upperBound) uncommon-trap
loop:
   if  int(i) <= int(upperBound)
   sum := int(sum) + int(i).
   if overflow(sum) then uncommon-trap
   i := int(i) + 1
   if overflow(i) then uncommon-trap
   goto loop
 else
   return sum)
```

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   if !hasIntegerTag(upperBound) uncommon-trap
loop:
   if  int(i) <= int(upperBound)
   sum := int(sum) + int(i).
   if overflow(sum) then uncommon-trap
   i := int(i) + 1
   if overflow(i) then uncommon-trap
   goto loop
else
   return sum)
```

The resulting n-method is as efficient as it can reasonably be, given that it is dynamically typed, and overflow-safe.

1 sumTo: 5

```
sumTo: upperBound = (| sum <- 0 |
   | i |
   i: self.
   if !hasIntegerTag(upperBound) uncommon-trap
loop:
   if  int(i) <= int(upperBound)
     sum := int(sum) + int(i).
     if overflow(sum) then uncommon-trap
     i := int(i) + 1
     if overflow(i) then uncommon-trap
     goto loop
   else
     return sum)
```

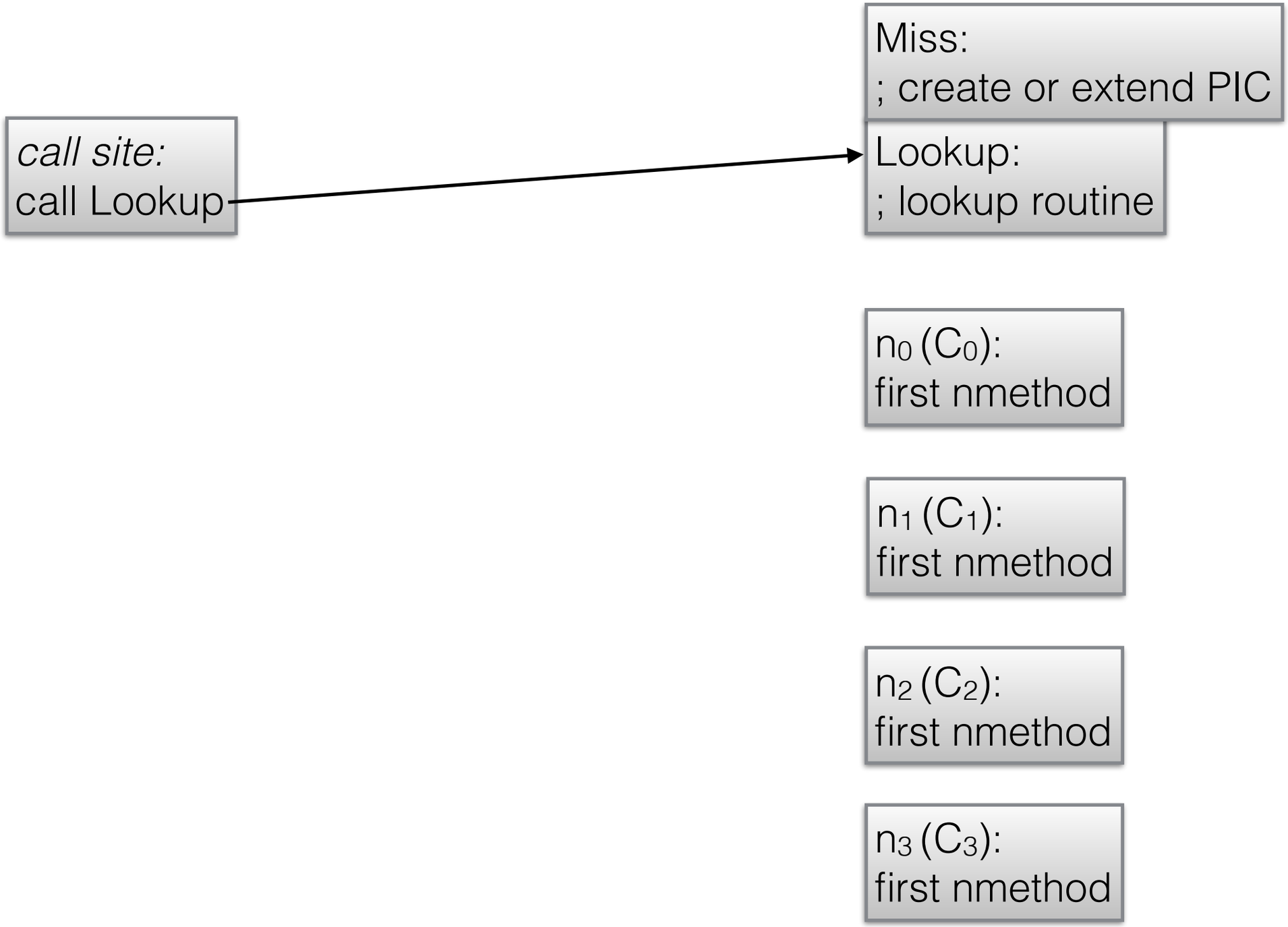
Example adapted from [Chambers and Ungar 1989], *Customization: Optimizing Compiler Technology for Self, a dynamically-typed object-oriented language*

The resulting n-method is as efficient as it can reasonably be, given that it is dynamically typed, and overflow-safe.

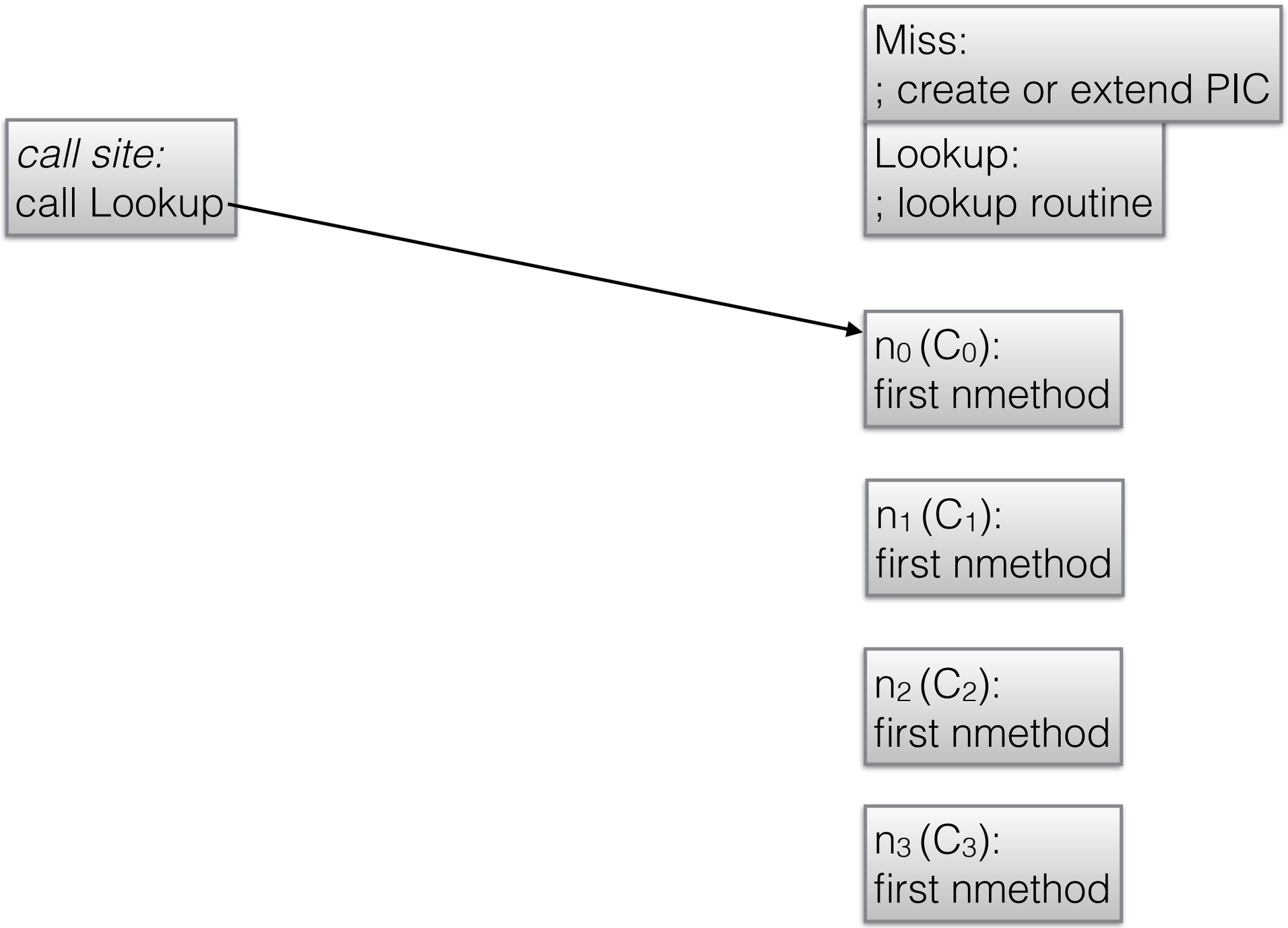
Self 3.0
(released 1993)

- Feedback-driven adaptive optimization (Urs Hölzle's thesis, 1994):
  - Polymorphic Inline Caches (PICs) and counters
  - Adaptive inlining
  - Deoptimization

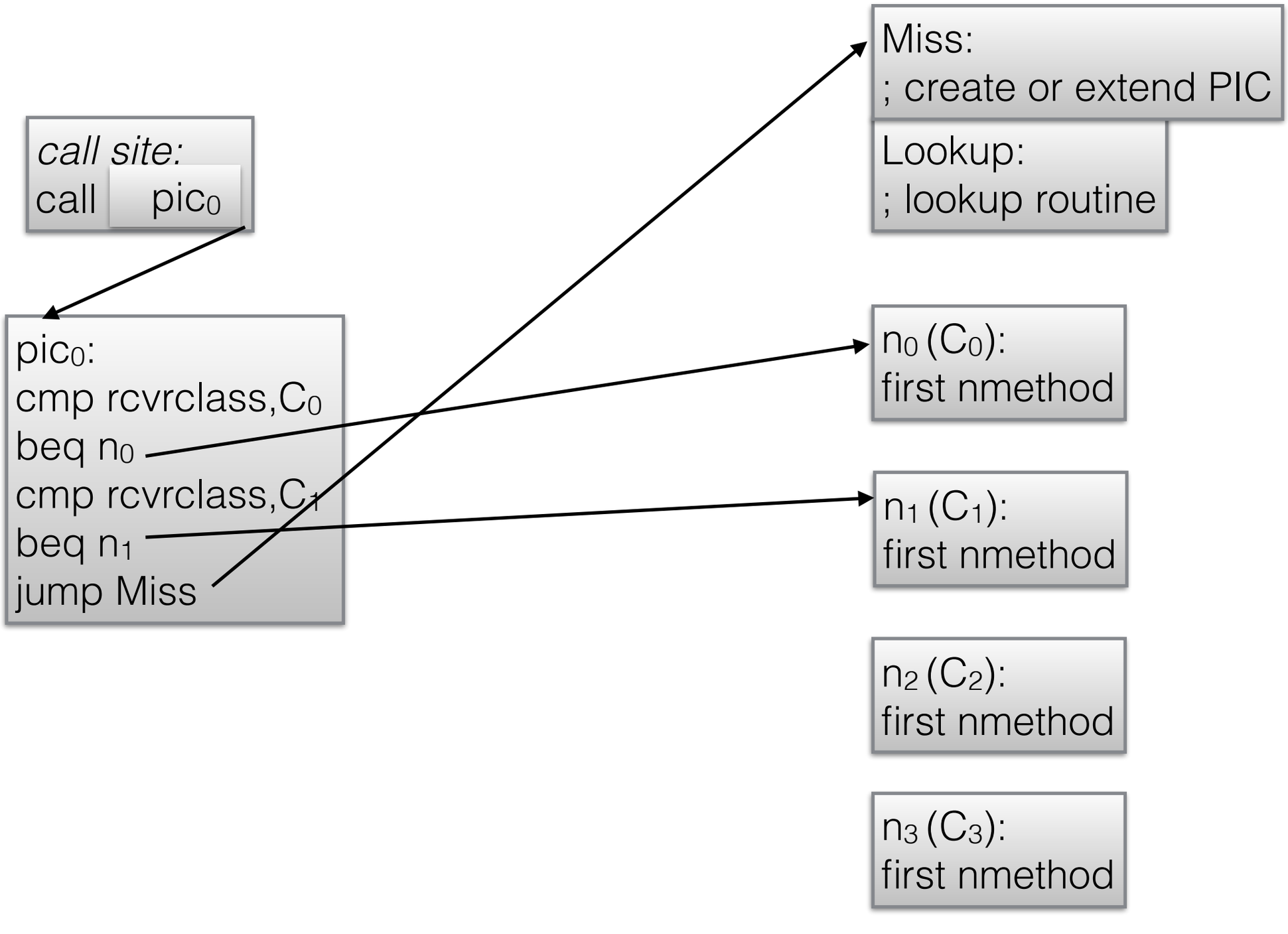
PICs — Polymorphic Inline Caches



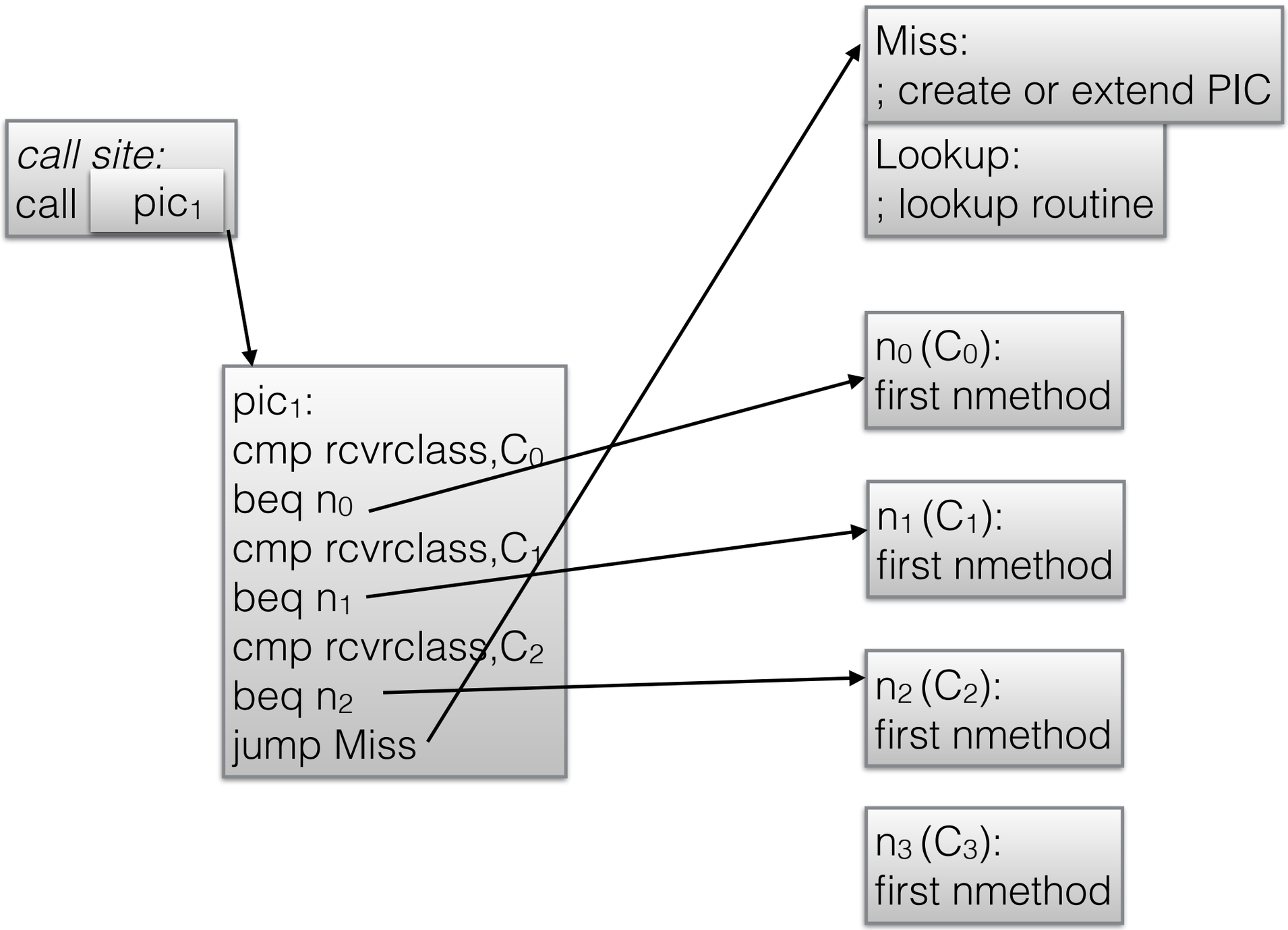
PICs — Polymorphic Inline Caches



PICs — Polymorphic Inline Caches



PICs — Polymorphic Inline Caches



PICs — Polymorphic Inline Caches

call site:
call pic_2

pic_2:
cmp rcvrclass,C_0
beq n_0
cmp rcvrclass,C_1
beq n_1
cmp rcvrclass,C_2
beq n_2
cmp rcvrclass,C_3
beq n_3
jump Miss

Miss:
; create or extend PIC

Lookup:
; lookup routine

n_0 (C_0):
first nmethod

n_1 (C_1):
first nmethod

n_2 (C_2):
first nmethod

n_3 (C_3):
first nmethod

Dynamic deoptimization

- Many potential optimizations are speculative: they are based on the current state of the program and/or data, which may change.
  - Example: Java class loading can invalidate a class hierarchy analysis

- If this occurs, we need a technique to recover the state of the computation, abandon the incorrect optimizations, and proceed with the correct behavior.

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

```
frame for m':
in call to bar()
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

frame for m': in call to bar()	frame for bar

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

frame for m': in call to bar()	frame for bar	fram

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

frame for m': in call to bar()	frame for bar	fram

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m’() {bar(); return 7;}
```

inlining tree

frame for m: in call to foo() x = 3 y = 4	frame for foo: in call to bar()	frame for bar	fram

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m'() {bar(); return 7;}
```

inlining tree

frame for m: in call to foo() x = 3 y = 4	frame for foo: in call to bar()	frame for bar: return to foo()	fram

Deoptimization safe points

- For speculative compilation, a general-purpose safety net is to have *safe points* (or *deopt points*) at which the entire state of the computation, as it would be represented in an interpreter, can be restored.

- Each deopt point is a value of the nPC that corresponds to a source program location (vPC or BCI), and which may have been inlined (transitively) into the current n-method

- Need to keep track of the stack of inlining points at each deopt point; induces a tree of *scope descriptors* (aka *frame states*).

- Within each descriptor we have a map of all source-level local variables and arguments and their locations (register/stack) or values (if constant).

- Keep copies of otherwise dead values, if required for debugging or reflection

On-stack replacement

Suppose we have a long-running loop. When we first execute the loop, all the methods are unoptimized. Part way through, we trigger a counter and invoke optimizing compilation of the loop and its callees. How do we transition to the optimized code before waiting for the loop to end?

Solution: use the same frame-replacement techniques, except this time replacing unoptimizing frames with their optimized counterparts.

Java

1995—present

VMs enter the mainstream

The Java Virtual Machine

- Java emerged shortly after the World Wide Web was invented; when dissatisfaction with C and especially C++ as an application language was high; and when OOP was hugely popular.

- Portable binaries + type-safe + objects

The JVM

- The JVM was based on familiar ideas: a machine-independent bytecode ISA; automatic memory management (GC); objects and methods.

- It added a class-level distribution format, sandbox security (applets), static typing and bytecode/class verification.

- Massive adoption made bytecode VMs and those implementation techniques ubiquitous.

JVM developments 1995–2000

- Early JVMs (1995-1998) were just playing catch-up with Smalltalk and Self.
- Many simple JIT compilers were written
- Java’s built-in concurrency added new challenges and opportunities

HotSpot

- The “Java HotSpot Virtual Machine” (1999), incorporated many of the techniques from Self…unsurprising, as developed by an ex-Selfer, following a pivot from Smalltalk:
    - Inlining, PICs, counters, depot
- Added new techniques for Java’s peculiarities, and careful engineering to take advantage of static types:
    - Fast locking, virtual table dispatch, …
- A subsequent release incorporated the Server Compiler (C2), which brought SSA-based heavy-duty code optimization techniques, taking performance well beyond that of Self-era compilers (such as the first HotSpot compiler, and the Client Compiler (C1)).

Later innovations used in JVM implementations

- Escape analysis
- Biased locking
- Thread-local allocation buffers
- Separable compiler(s)
- Concurrent GC
  - Lots of techniques

VMs for small devices

- The language VMs of the 1960s and 70s had run on machines with much less than a megabyte, but the adoption of dynamic compilation as the central approach (since PS) had increased memory consumption dramatically.

- In the late 1990s, new, resource-constrained platforms were emerging (PDAs, cellphones) which could not accommodate desktop and server VMs.

- There was a need to go back to earlier approaches.

Spotless and the K VM

- In 1997–8, Antero Taivalsaari, Bill Bush and Doug Simon at Sun Labs developed Spotless, a cut-down JVM for the Palm PDA (128KB)

- This became the K VM, the JVM of J2ME CLDC and shipped on hundreds of millions of cellphones.

- The next CLDC JVM HotSpot Implementation, aka “Monty”, could afford to adopt dynamic compilation again — phones had more memory.

- The Exact VM — which had vied with HotSpot on the server — eventually was repurposed as the CDC JVM (BluRay and elsewhere)

- There were always be a market for small VMs running on tiny devices; but the market has changed many times.

Microsoft Common Language Runtime

- The first VM intentionally multilingual(?)
  - many previous attempts at hosting other languages on Smalltalk, Self, Java VMs.
  - Managed/unmanaged
- C#, C++, F#, J#, JScript, P#, Visual Basic, Iron*, …

System VMs

Part 2, 1975–2000

Recap…

- 1960s: invention of the System VM, adopted by mainframe users

- 1970s: proliferation of VMs in the mainframe world. Ignored by academia and minicomputing.

Co-designed VMs: Hardware and VMs designed together

- In the 1970s, one IBM division took mainframe VM technology a step further by separating the guest instruction set from the native ISA, rather like language VMs had done.

- IBM System/38 (1979) included:
    - A virtual ISA (translated to native ISA ahead of time, but *not* at development time). Applications were distributed using the vISA. The nISA was not visible to users (except for the act of translation, which was opaque).

    - A higher-level machine model using objects and capabilities. Objects were supported by the OS, file system, etc.

- The product line evolved into the AS/400 (1988), iSeries (2000), System i (2006); rolled into Power Systems (2008) with “IBM i” OS. Commercially successful.

The 1990s: System VMs can solve a new problem

- During the late 1970s and 1980s the computing economy had moved from mainframes to minicomputers, and in the late 1980s and 1990s, microprocessors began to dominate.

- By the late 1990s a data center might have had hundreds or thousands of microprocessor-based computers, many of which were idle at any given moment. Why?

The dirty secret of computing

The dirty secret of computing

Any real system is a composition of many software packages, and

every specific configuration of package versions has to be tested

to verify correctness (in a pragmatic, not absolute sense).

Version proliferation

- The result was that in a data center running many applications, each application often ran on a dedicated computer, and large-scale applications had as many computers provisioned as were needed for peak demand.

- Hence, many—most?—machines were underutilized.

The System VM Renaissance: x86 virtualization

- Begun by VMware in the late 1990s
- Solution: every application stack runs on a System VM. Each computer can host many such VMs.
- Business model: some fraction of the money saved on computers can be spent on the VMM.
- Bonus: An attractive solution for development, QA of multiple versions (Windows*, Linux, …)
- Problem: x86 does not meet the Popek-Goldberg requirements. Solution: Dynamic binary translation

Binary Translation, Process VMs and ISA-Translating System VMs

Dynamic binary translation

- What is it?
  Translation of machine code from one ISA to another at runtime.

- Why dynamic?
  In the worst case, the application may generate code at runtime which a static translator will never see.

- What's it good for?
  Executing programs compiled for one ISA on another; simulation (translations that gather/model additional state); dynamic binary optimization

- Early examples: MIMIC, Shade, DAISY, Mac 68K emulator, Dynamite.

The *Dynamo* binary code *optimizer*

- Ran HP PA-8000 PA-RISC applications, dynamically re-optimizing hot code to improve performance (same input and output ISA)

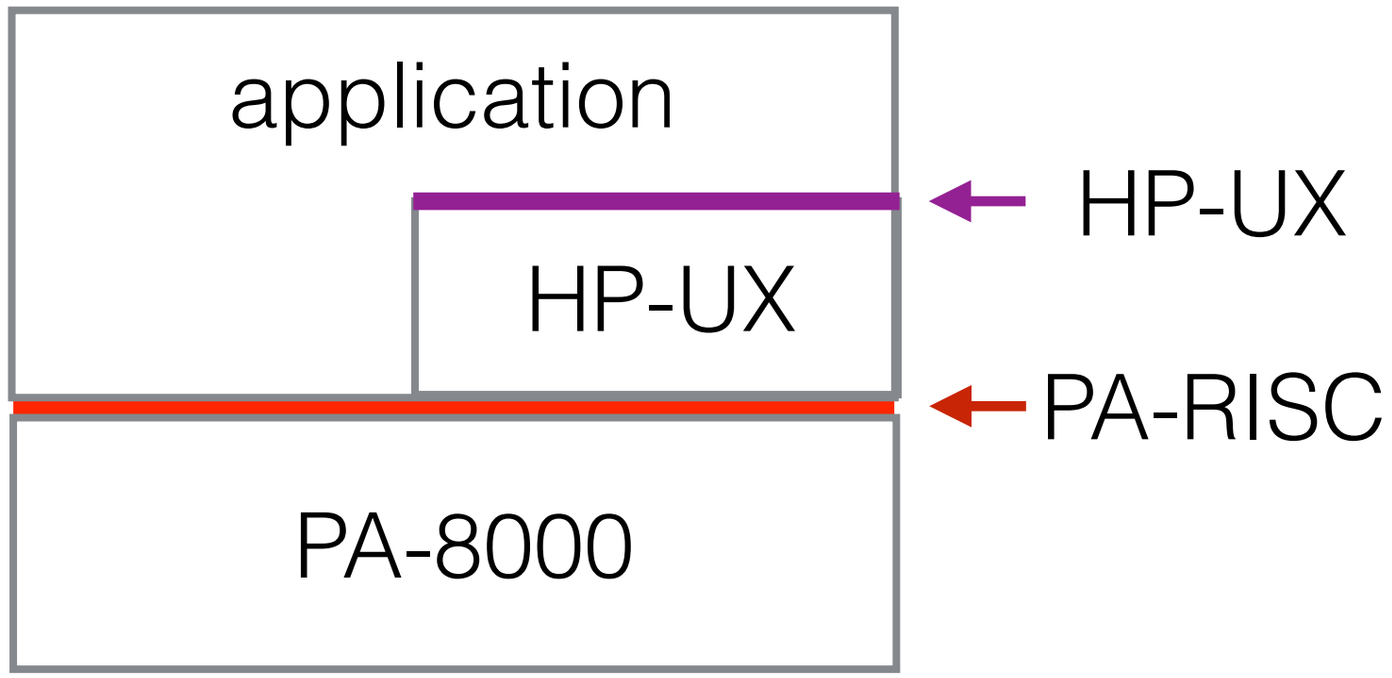
- An example of a **trace-driven** translator

- The unit of optimization is an *instruction trace* (can span many basic blocks and procedures)

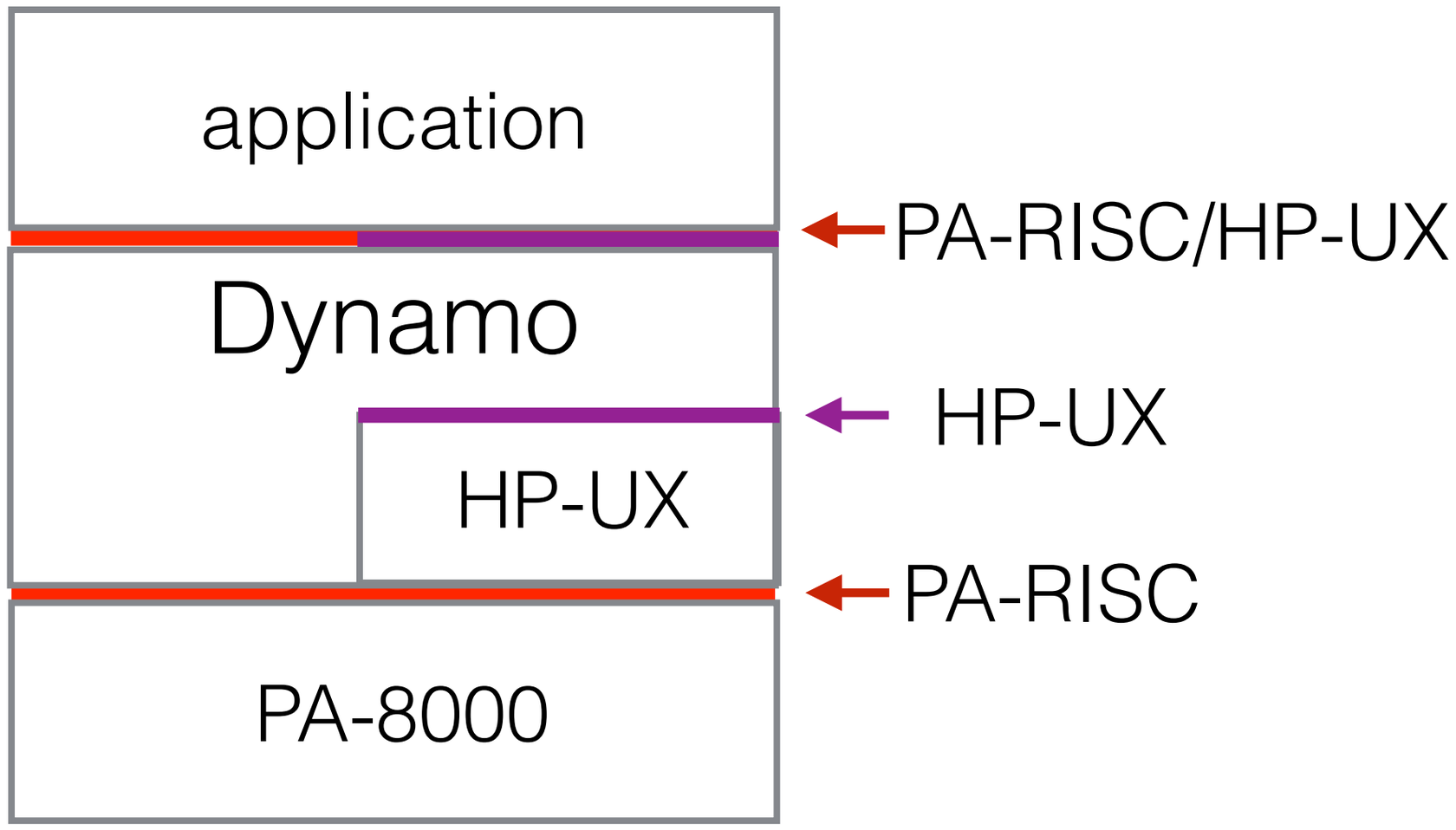
- A **Process VM** (implements the ABI)

- *Dynamo: a transparent dynamic optimization system*, Bala et al., PLDI 2000

Dynamo



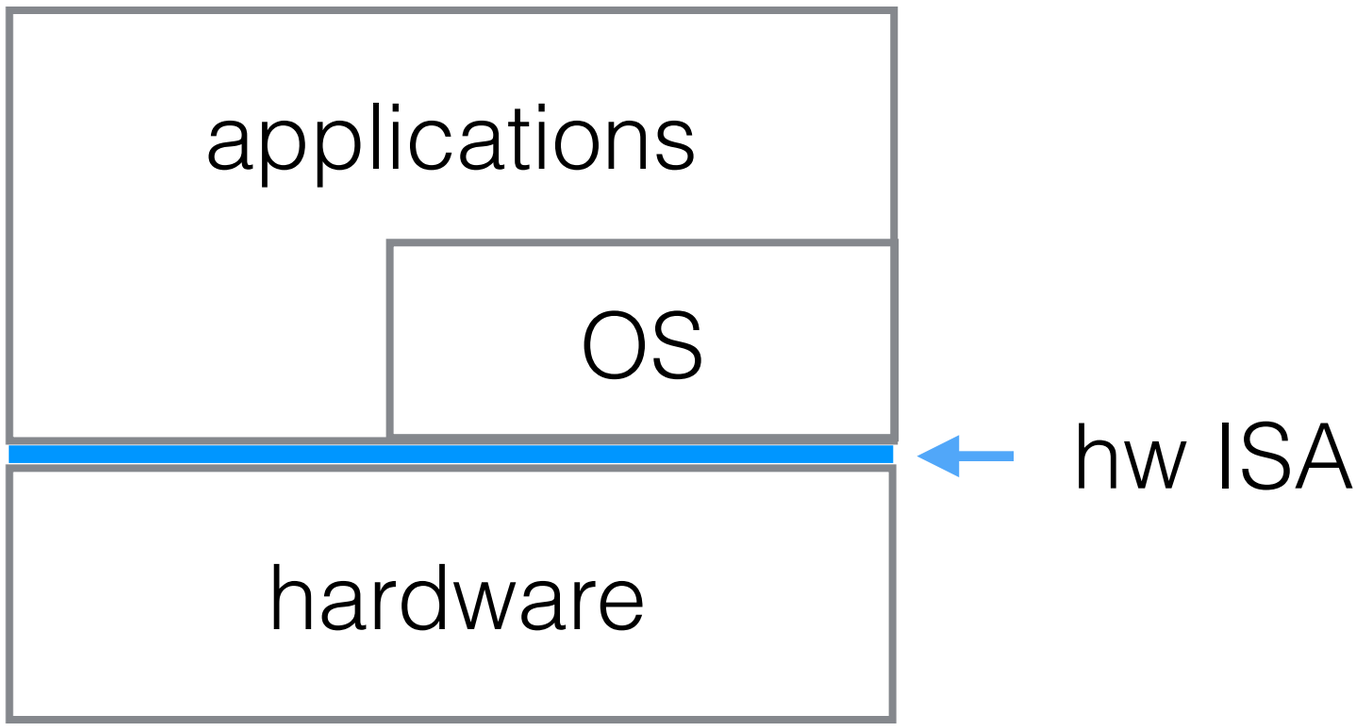
Dynamo



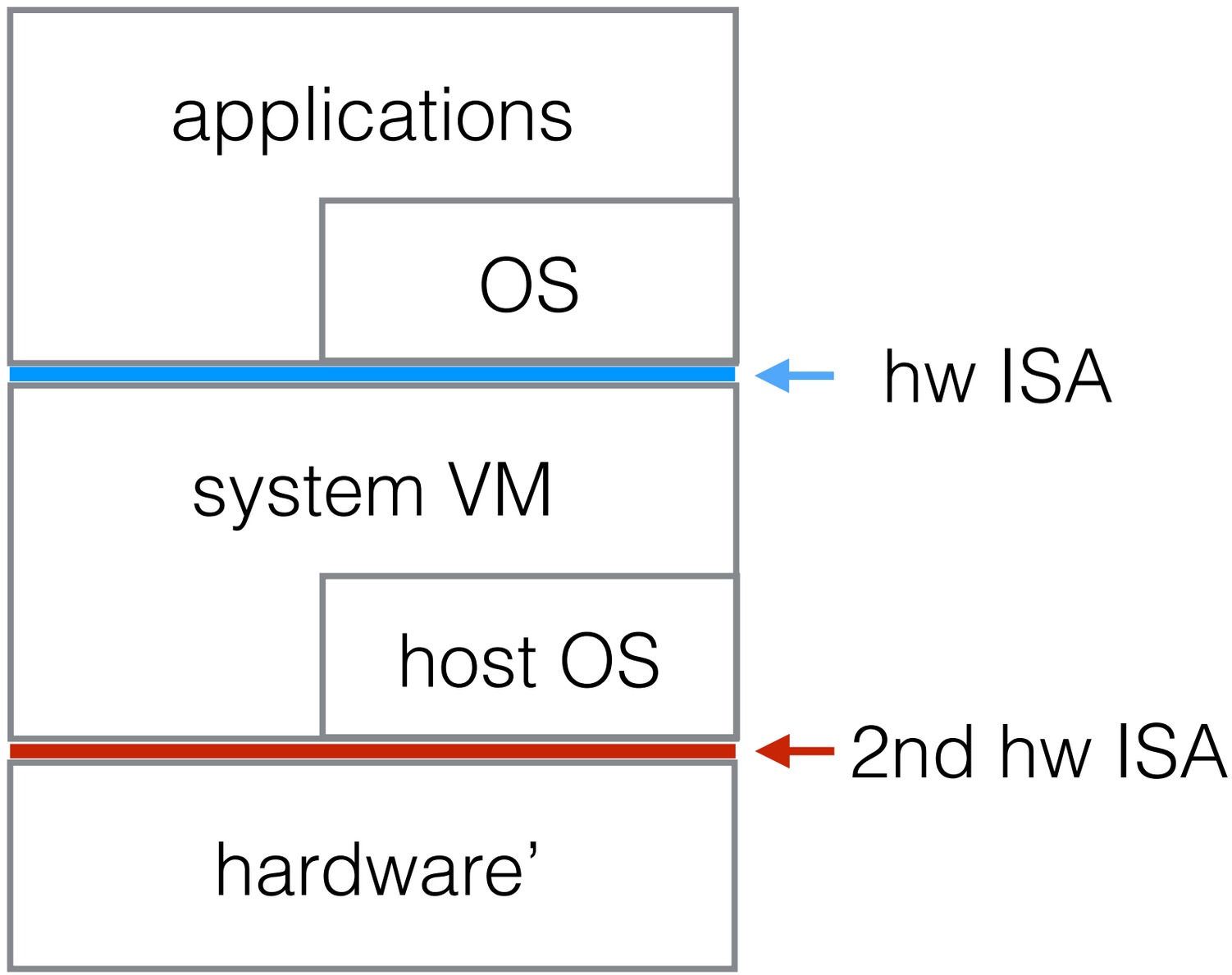
Process and System VMs

- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)
- A **System VM** implements *both* the hardware user and system ISA
  - A **Hosted System VM** has a host OS
  - A **Classic System VM** has a Virtual Machine Monitor (not a full OS; you can't run applications on the VMM as it does not implement the ABI)

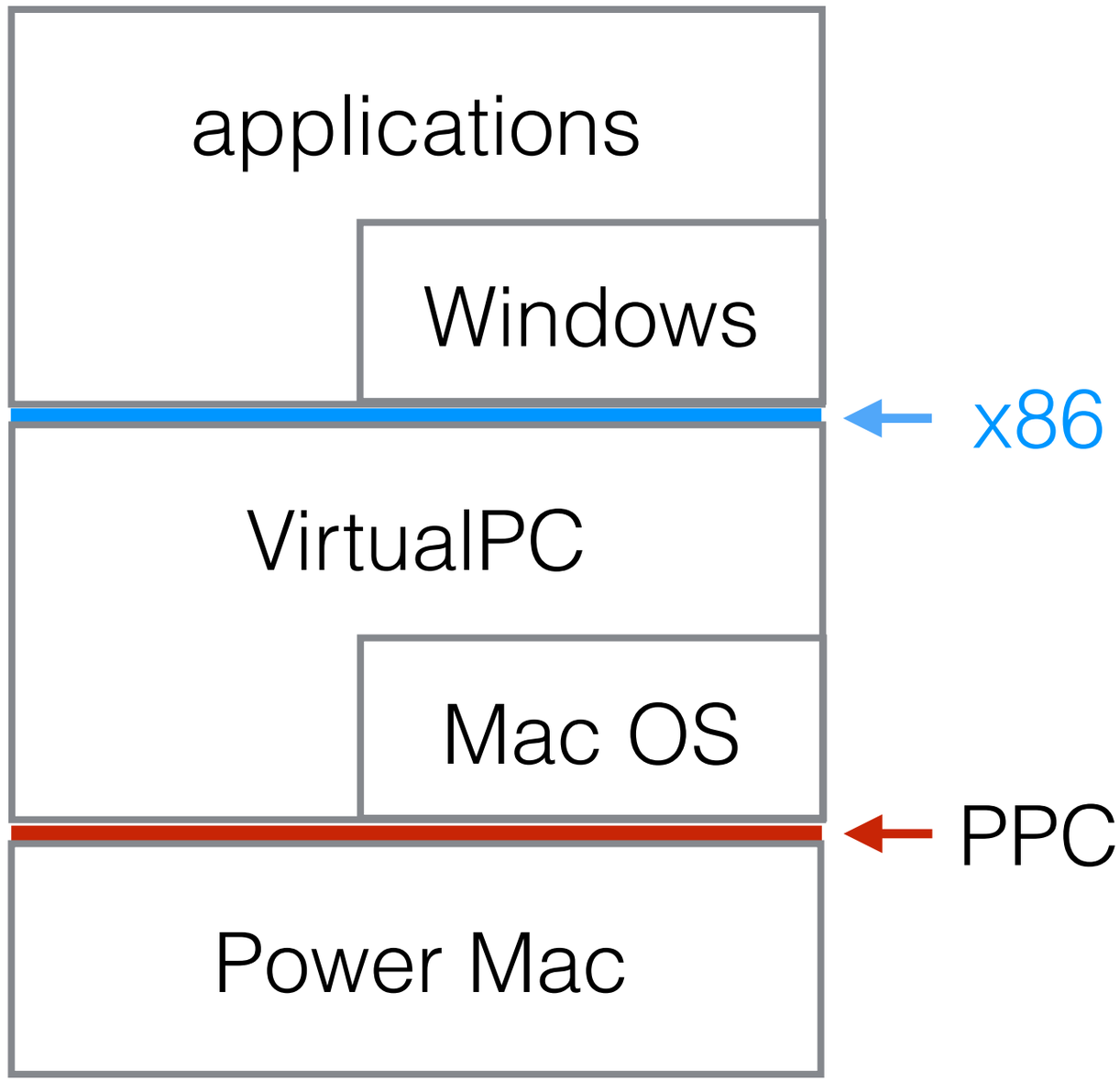
Architecture of Hosted System VMs



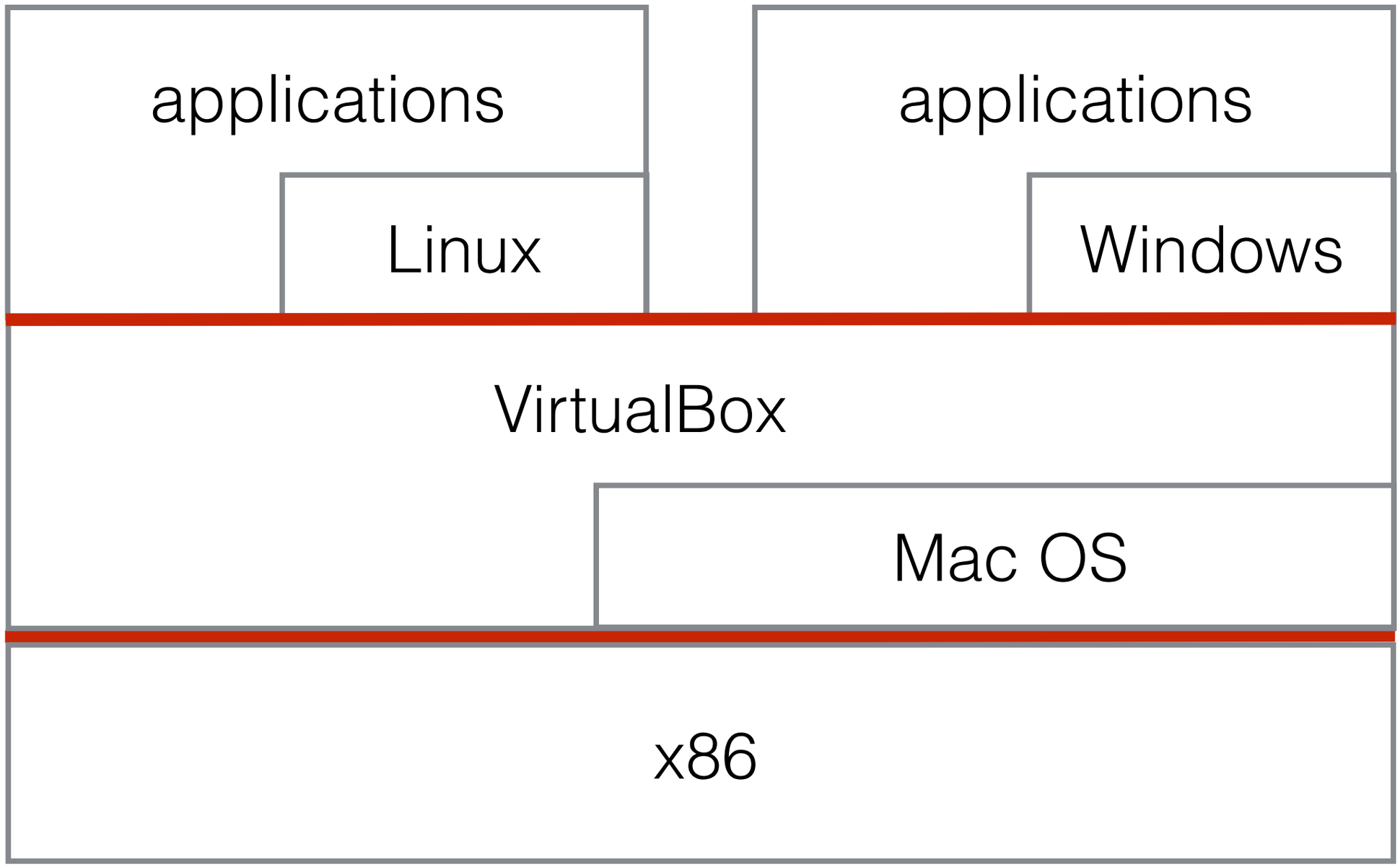
Architecture of Hosted System VMs



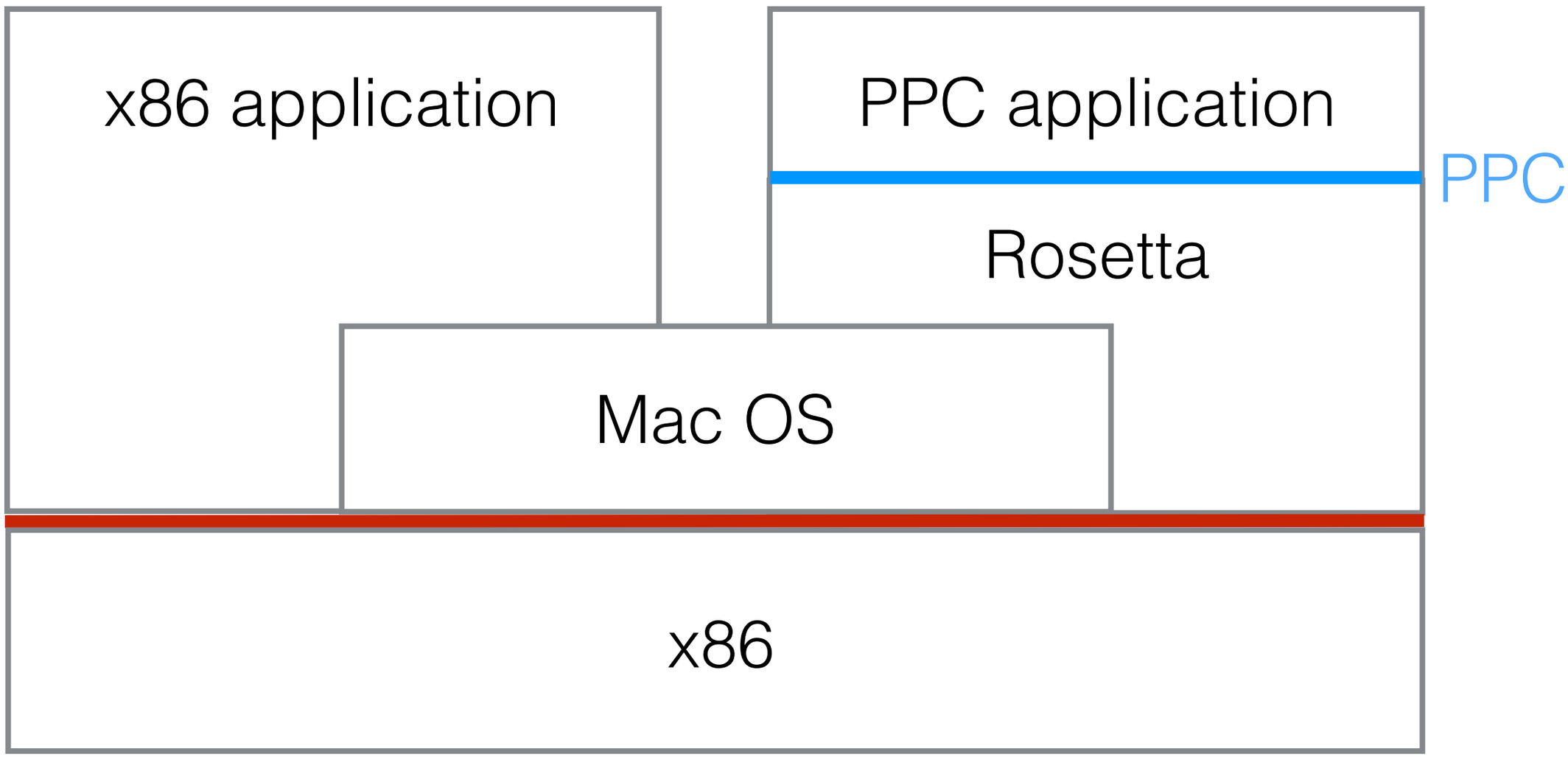
A Whole-System VM: *Virtual PC* (1997)



VirtualBox example



Rosetta architecture



How it worked

- When a PowerPC binary was invoked, the Rosetta layer performed a combination of interpretation of PPC instructions and dynamic binary translation (translating chunks of PPC code to x86) to run the application

- Because the new x86 processors performed better than the previous-generation PPC processors, the resulting performance was comparable, and therefore “good enough” until the applications were available as x86 binaries

- This was the *second* such system deployed by Apple: they used a dynamic binary translator when moving from 68K to PPC (Wikipedia: Mac_68k_emulator).

Transmeta's *Crusoe*

- **Co-designed** software and hardware VM that ran IA-32 code on a VLIW-architecture microprocessor via a **dynamic binary translator**.

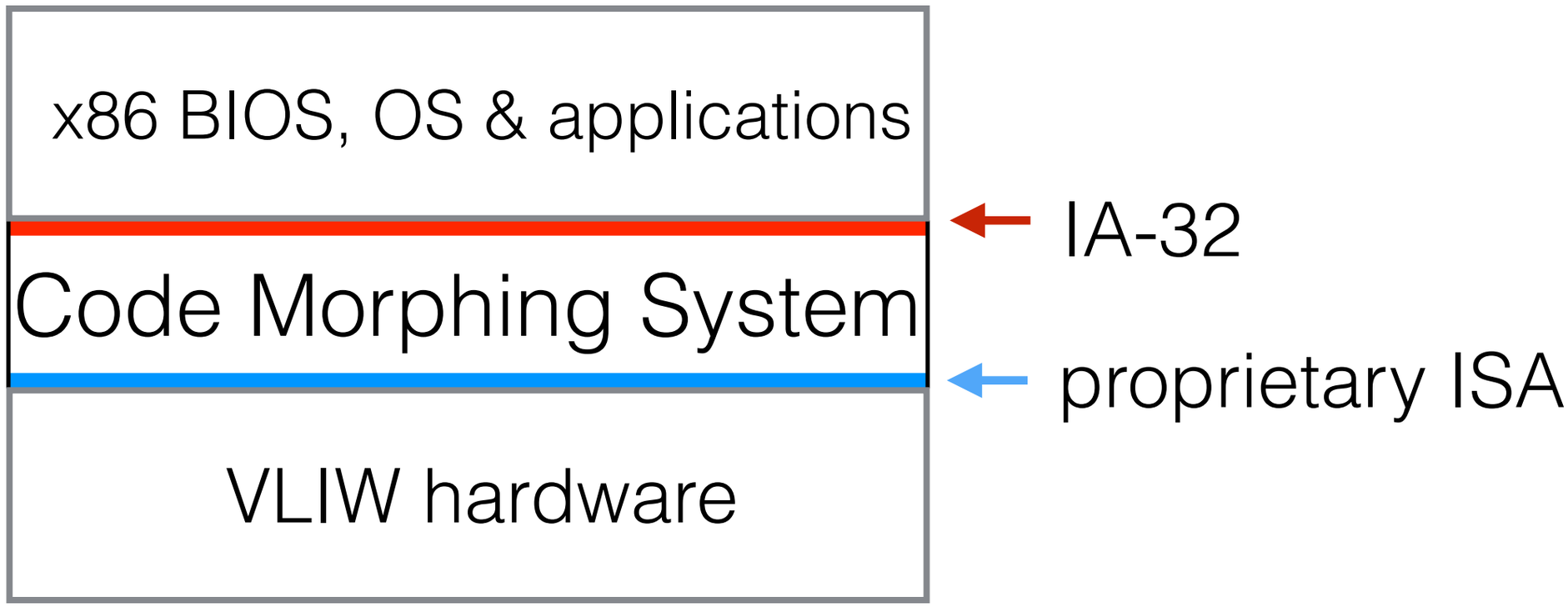
- Goal was improved power-efficiency with performance comparable to a conventional x86 implementation

    - Achieved by eliminating power-hungry hardware, such as fast x86 instruction decode

- Used in a variety of laptops, tablets, notebooks

- Introduced in 2000; a follow-on (Efficeon) in 2003

Crusoe system architecture



- CMS used many of the techniques from Language VMs (feedback-driven optimization, speculation)
- Hardware provided additional support to deal with speculation and translation (e.g., shadow registers and gated store buffer with rollback triggered by alias detection)

System VMs

Part 3, 2000 to the present

Paravirtualization

- Traditional system VMs implement not only the user and privileged instruction sets, but must also emulate devices.
- In a paravirtualized VM, more abstract devices are implemented, which the VM maps onto actual devices
  - Simpler, more efficient
  - A modest OS porting effort is required
- *Xen and the Art of Virtualization*, Barham et al., 2003.

Hardware virtualization comes to x86

- In the mid-2000s, system virtualization was becoming so popular that the chip manufacturers decided to clean up their architectures and support virtualization directly.

- AMD-V, Intel VT-x for x64

- Many other ISAs also took this step in that decade (e.g., SPARC, POWER, ARM)

Cloud — the killer app of the system VM

- System virtualization is the basis of elastic cloud computing, and its most famous embodiment, Amazon EC2 (Elastic Compute Cloud) — based on Xen

Nested virtualization comes of age

- Mentioned by Popek and Goldberg in their 1974 paper, it took four more decades for nested virtualization to become mainstream.

- Haswell (2013) introduced hardware support for nested virtualization.

- Ravello Systems acquired by Oracle for M$500 (2016) — runs VMware VMs on EC2 or Google Clound; based on binary translation.

Universal ISAs/IRs

- UNCOL (1958)
- ANDF — Architecture Neutral Distribution Format (1989) call for proposals,
- **LLVM** 2003—present
- Easy if your universe is finite, small, and fully explored! Otherwise…

Language VMs

part 2, circa 2000 to the present

Proliferation

- By the mid-2000s, language VM technology had been widely deployed (on perhaps a billion devices, from cellphones to supercomputers)

- The bulk of the implementation effort had gone into JVMs (Sun, IBM) and the CLR (Microsoft).

- In contrast, the performance of other managed languages (JavaScript, Python, Perl, etc.) was lackluster.

JavaScript Wars

- Language was invented by Eich at Netscape in 1995
- By mid-2000s, it was still the only viable language of the web, but was interpreted.
    - OK for web-page one-liners, not for web applications. AJAX made sophisticated applications possible.
- In the late 2000s, several companies invested heavily to develop high-performance JavaScript VMs:
    - Mozilla: TraceMonkey — trace compilation comes to language VMs
    - Google: V8 — very similar to Self (maps) and HotSpot
    - Apple: WebKit/SquirrelFish (later Nitro)

Trace compilation

- In a binary translator, traces are a more obvious choice for translation unit. In a language VM, the linguistic constructs are available — so why use traces?
- Linear traces are easy to compile quickly
- Inlining comes for “free” — traces span call boundaries
- Well-described by Gal et al., HotPathVM, 2006 (a JVM)
- Used in Mozilla’s TraceMonkey

Making VMs easier to build

- Using a higher-level language
  - Jikes, Squeak, Squawk
  - J9 - Smalltalk, modularity
- Metacircularity
  - Klein, Maxine
- Metatracing (PyPy)
- Partial evaluation (Truffle)

Some disadvantages of writing a VM in C/C++

- Lack of safety
  - Useful in some places (e.g., GC) — but a hindrance elsewhere (e.g., when writing compilers)
  - Reliance on unspecified behavior
- Black-box, static compilation
  - Must manually record locations of oops for GC, no safe points
  - Can’t generate code at run-time from C/C++
- Adapting between calling conventions
  - stack overflow checks, float/int, others mentioned by Cliff Click
- Memory model mismatches
- Missing features (not low-level enough in some cases [eg threaded code, inline caches])
- Result: building a high-performance VM in C/C++ requires extraordinary skill and great effort.

Writing a VM in a high(er)-level language

- These issues have led to attempts to write VMs in other languages, to decrease the skill and effort level required. Desiderata:
  - Higher-level (e.g., type- and memory-safe);
  - Better low-level control when needed (to avoid assembly)
  - Uniform and preferably automatic handling of oops, safe points, calling conventions, etc.

If you can't beat them… Compiling to C

- An alternative approach is to use C as a backend implementation language
- Examples: Squeak (Smalltalk), Squawk (Java)
- Leverage the universality of C compilers
- Compiler to C written in (subset of) HLL
- Doesn't address some of the low-level issues (C convention, fine-tuning instruction sequences)
    - But translation can deal with oop-tracking; no need to worry about it in VM source
- C compilers are typically too slow to be used at run-time; not useful as a dynamic compiler
    - LLVM is challenging this position, with mixed results

Squeak overview

- Squeak is a Smalltalk system developed in the late 1990s; several of the original Xerox PARC Smalltalk pioneers were involved.
- The Squeak VM is implemented in a subset of Smalltalk, *Slang*
- The VM definition is based on the reference definition in the Smalltalk “Blue Book”
    - Can be run directly within Smalltalk for debugging and experimentation.
    - Slang is a subset of Smalltalk which is straightforwardly translated to C, and the compiled to make a new VM.
- Squeak JIT compilers, added later, do not go via C.

Metacircularity — with performance

To get performance together with the benefits of a higher-level language, we can adopt an architecture in which a single compiler can serve to build the VM and also to compile applications.

Metacircularity — with performance

compiler source

bytecode compiler
(e.g., javac)

preexisting VM

Metacircularity — with performance

compiler source

bytecode compiler
(e.g., javac)

preexisting VM

compiler bytecode

Metacircularity — with performance

compiler source

compiler bytecode

preexisting VM

Metacircularity — with performance

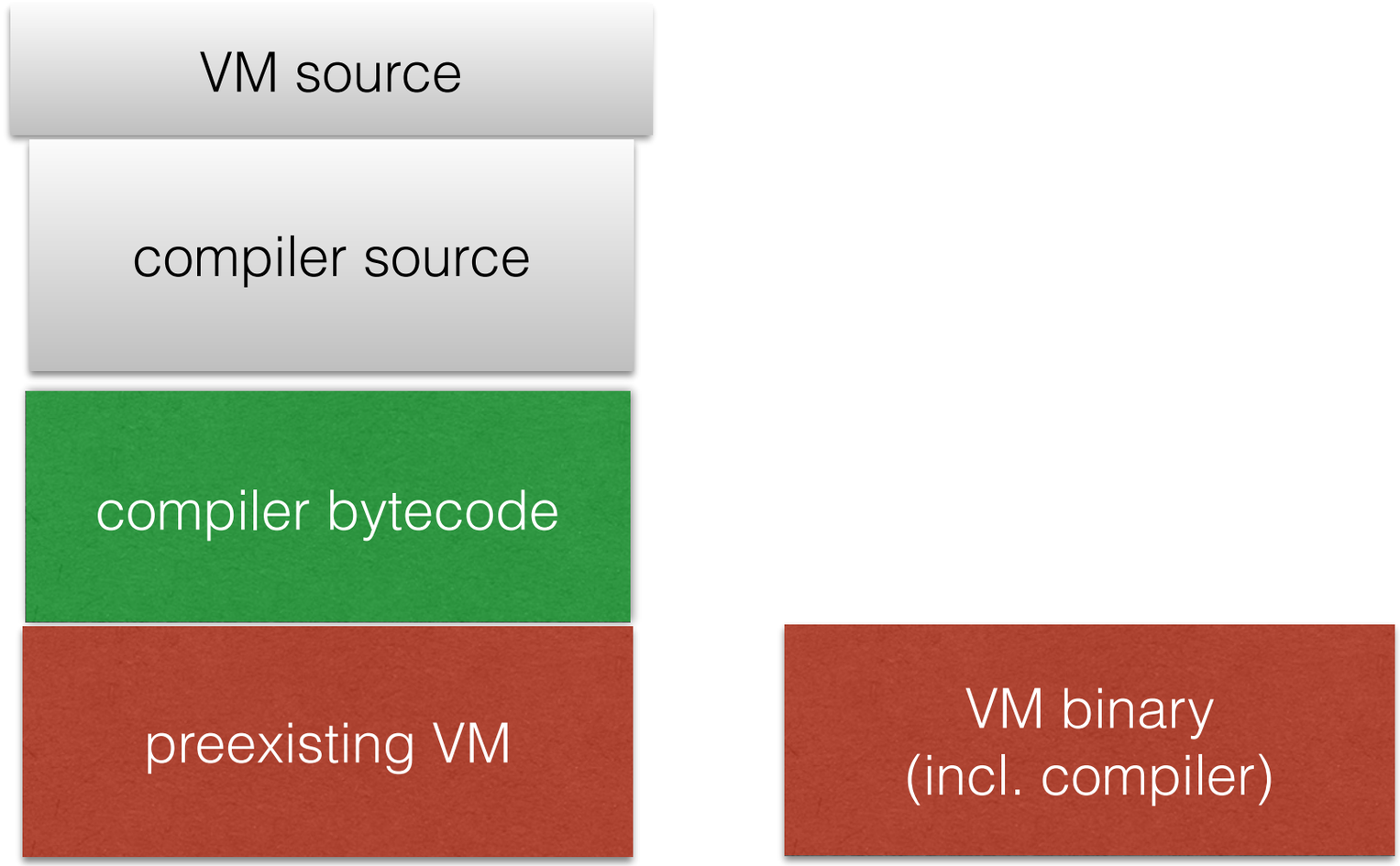
VM source

compiler source

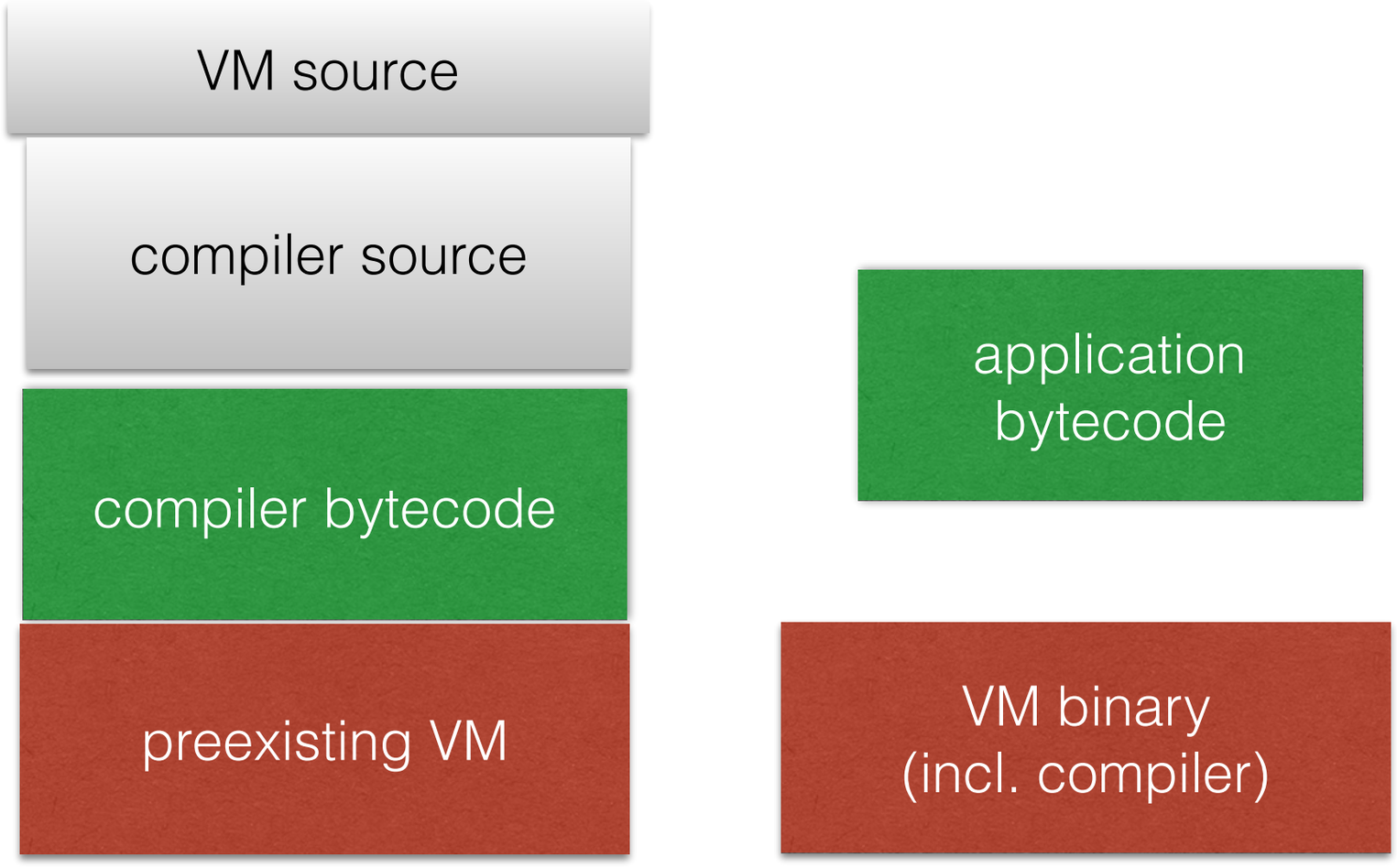
compiler bytecode

preexisting VM

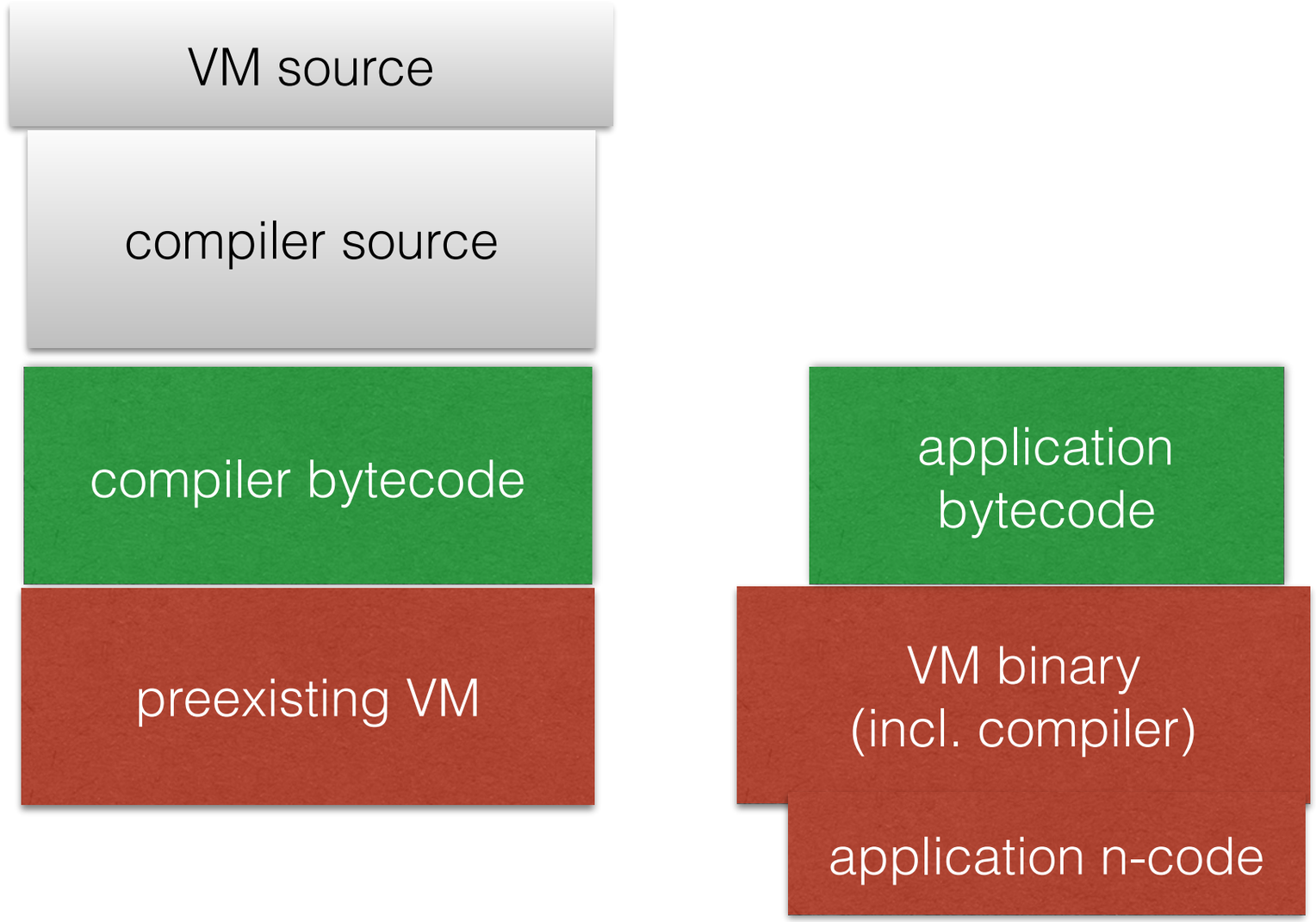
Metacircularity — with performance



Metacircularity — with performance



Metacircularity — with performance



Advantages

The same compiler is being used for the VM as for the application

- Common calling convention
- Can inline VM code into application
- Common handling of safe points, oops

Other observations

- Must be able to write a runtime (including GC) — need unsafe language features to manipulate memory
- Writing a GC without invoking GC is *tricky*.

Jikes Research VM (known initially as Jalapeño)

- Developed by IBM in the late 1990s as a research Java VM

- Open sourced and widely adopted in academia; hundreds of papers have used it as infrastructure

- Written in Java, extended with “magic”

Jikes Research VM

- Includes a rich and flexible GC system, MMTk (Memory Management Toolkit); port contributed by academia
- Includes 2 compilers (at least) — a baseline JIT compiler, and an optimizing compiler; no interpreter
- Hotspots are detected and compiled with an optimizing compiler, adaptively, using deoptimization and on-stack replacement
- VM builder constructs boot image with initial heap
    - Tree-shaking eliminates bloat, enables static optimizations

Magic extensions

- Extensions are easy when you're in charge of the compiler!
- Add machine-level data types (words, pointers, etc.)
- Add intrinsic methods for low-level access
- Wrap the above with types and annotations to make intended usage (and non-usage) clear and checkable.
- Other annotations can be used to drive inlining, bootstrapping, exclude GC, etc.
- Magic features unavailable to applications (different compiler mode)

The Klein VM

- A Self VM written in Self (not a subset)

- An exporter takes Self objects in the source world and writes out the bits they represent into the Klein boot image

- Mirror-based debugging is used to debug a remote Klein world from a Self world (mirrors are a kind of proxy and were included in Self for reflective operations; Klein extends them to work on a remote object)

The Maxine VM

- Maxine is also a JVM written in Java
- Modular Architecture
- Novelties:
  - T1X template JIT compiler
  - Snippets (high-level approach to IR weaving)
  - Inspector, uses JavaInJava for single step (see videos)

The Maxine inspector

- The Maxine inspector is a special-purpose debugger/ observer used in developing Maxine.

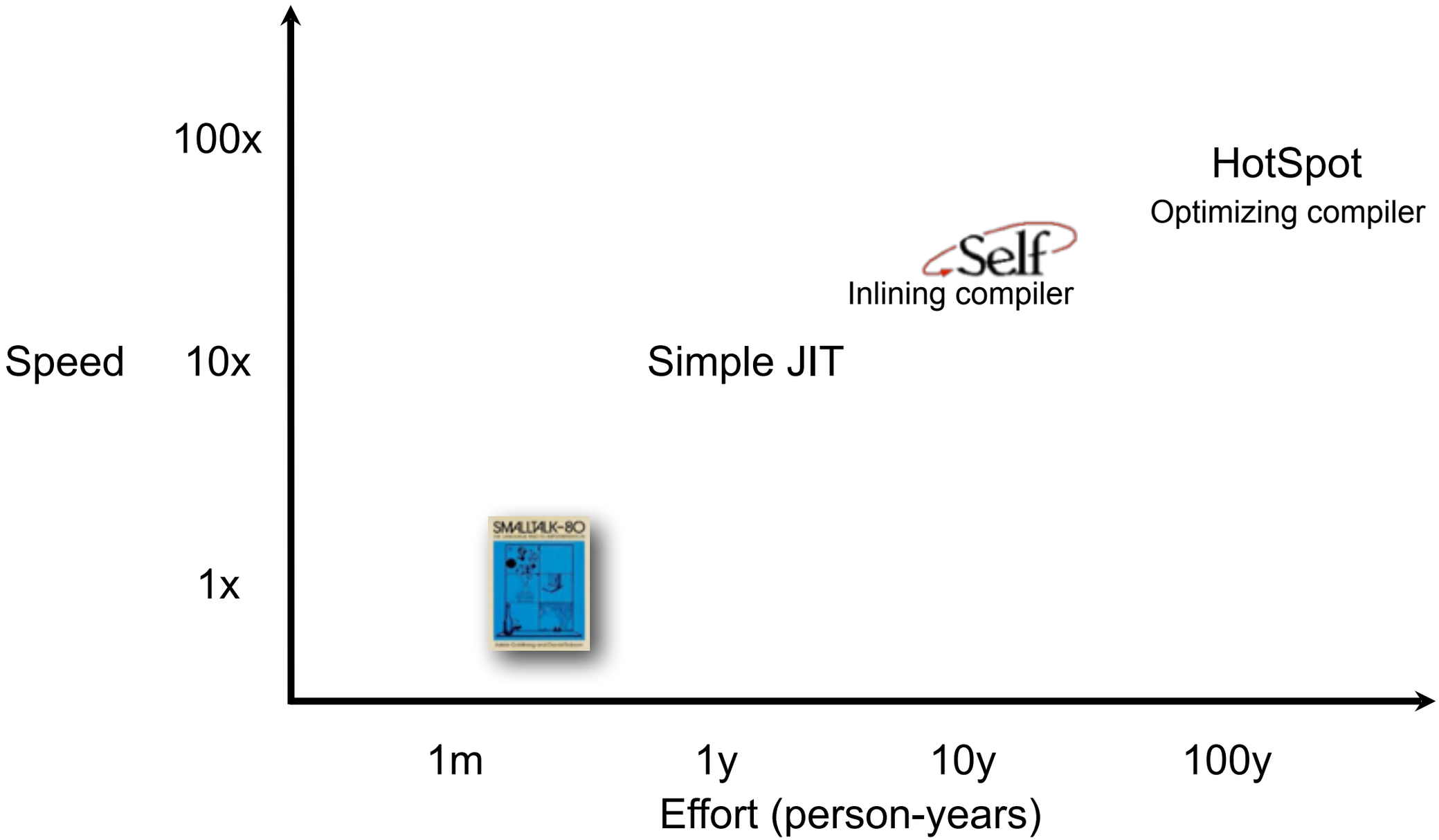
- Uses the metadata gathered during a build to be able to attach, observe, debug and control a Maxine instance — even if the instance is broken.

- Can interpret memory, registers, stack frames, objects, etc., to present a meaningful view to the developer.

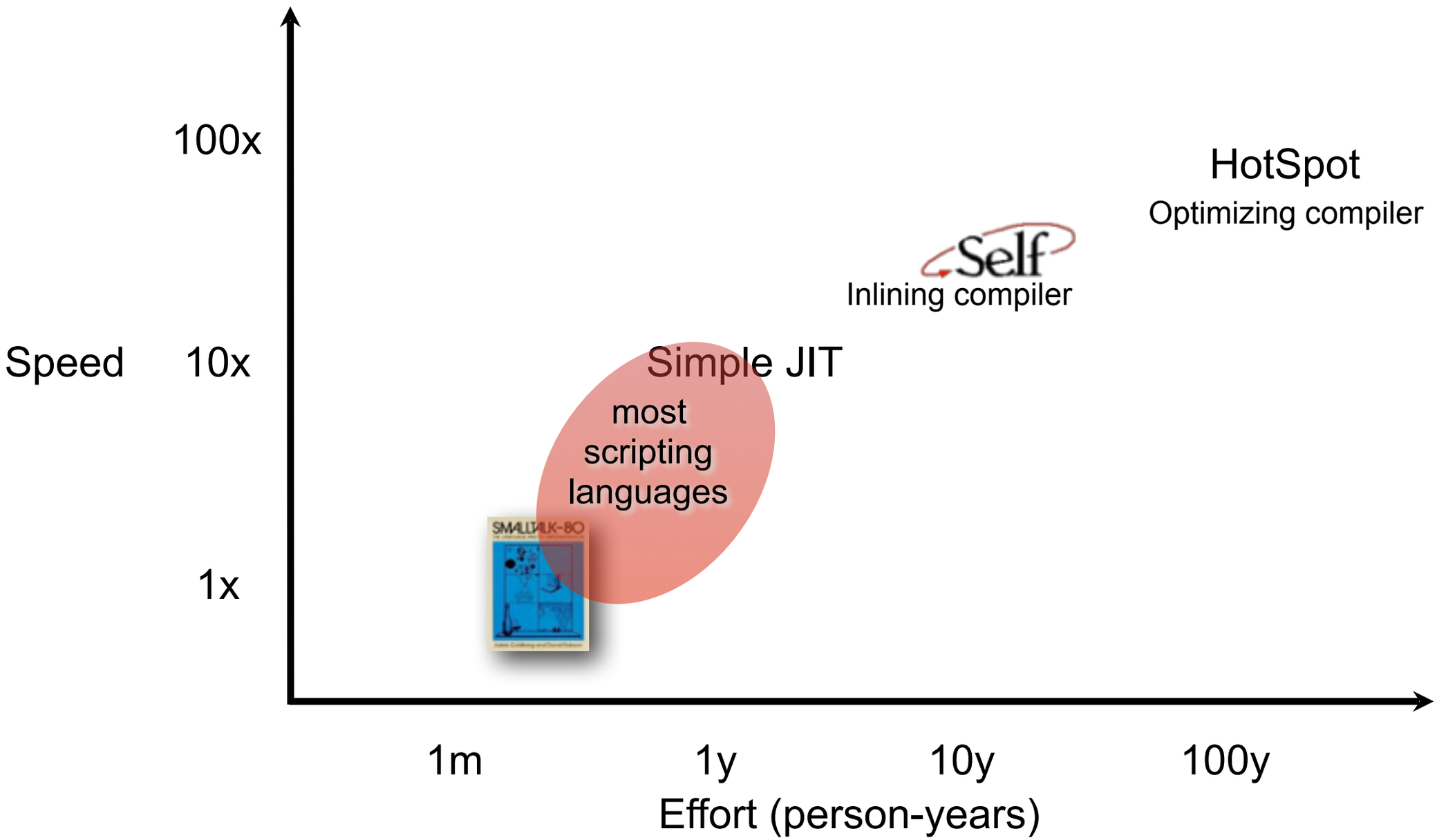
- Best seen in demonstration (youtube link to follow).

Multilingual VM Frameworks

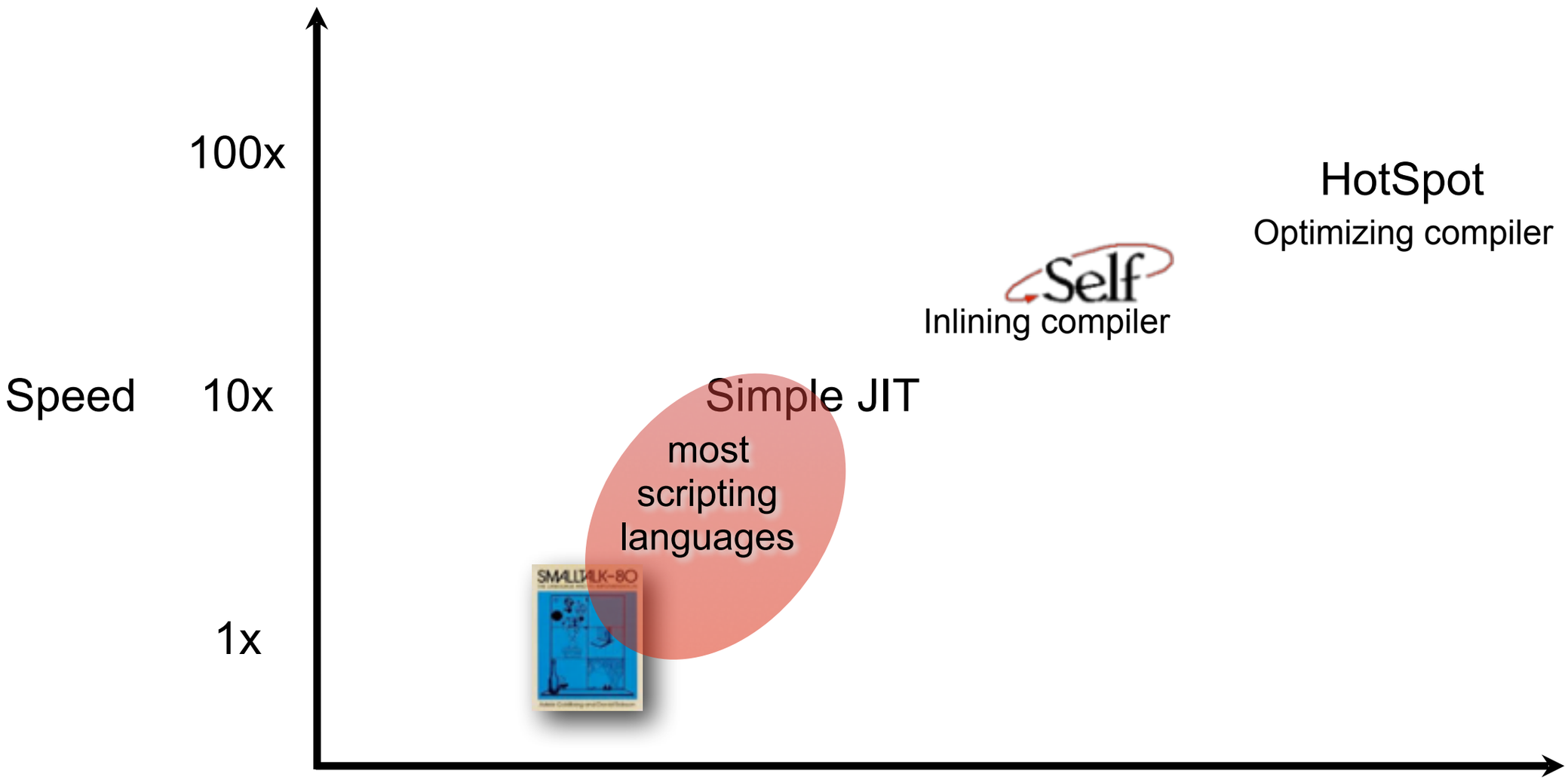
Building fast VMs is a lot of work



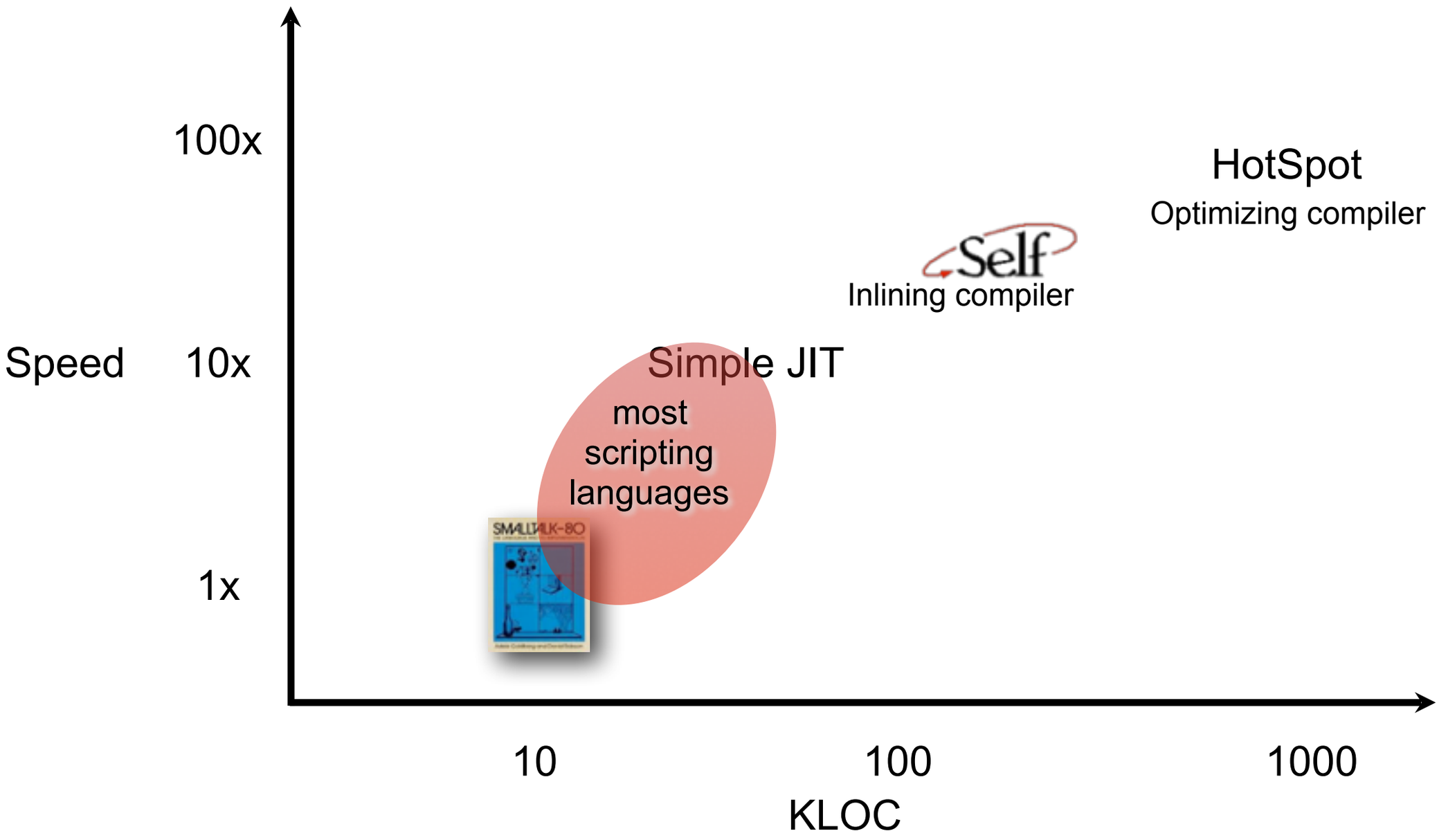
Building fast VMs is a lot of work



Building fast VMs is a lot of work

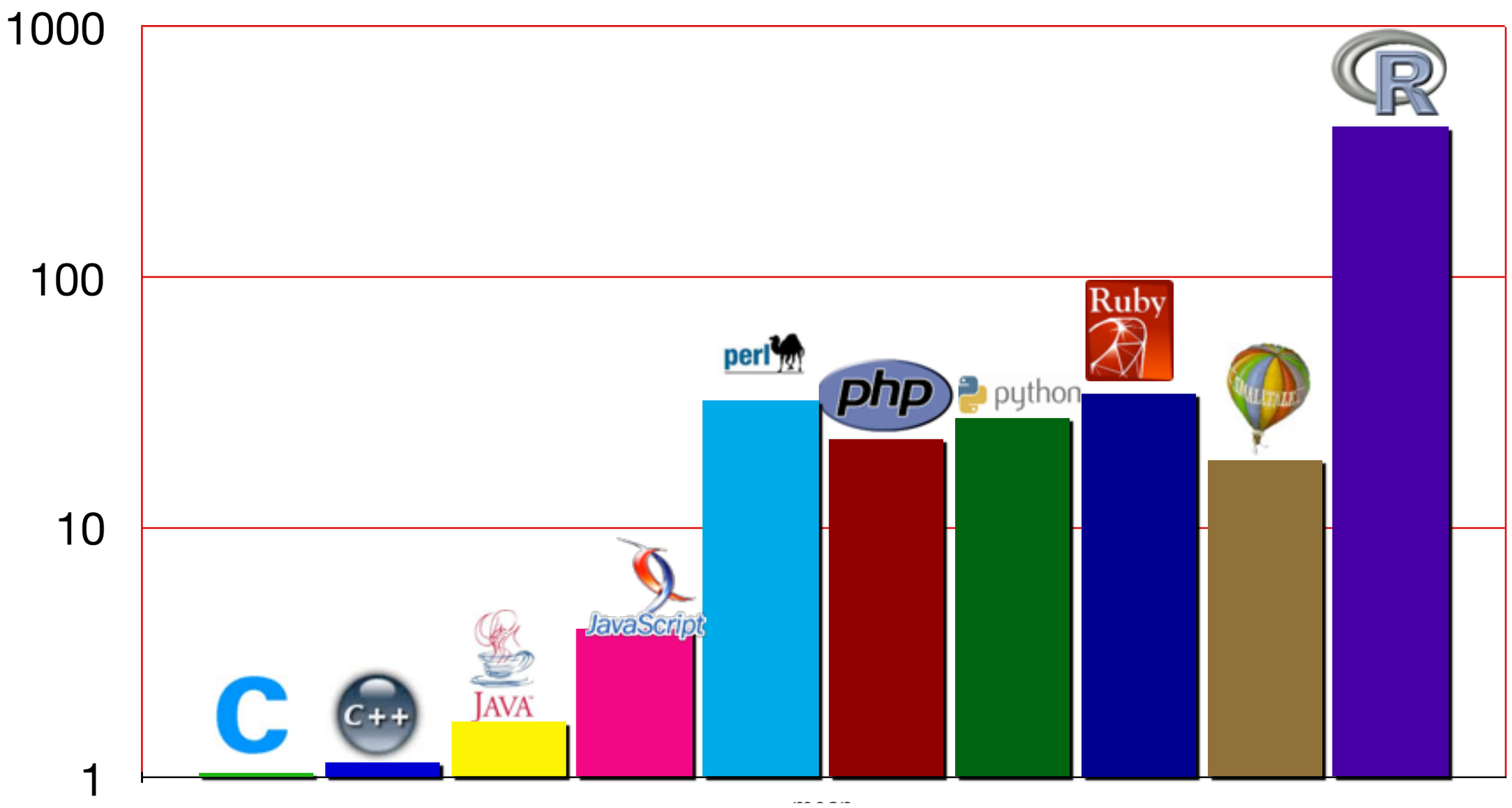


Building fast VMs is a lot of work



Relative speeds of various languages

From the Computer Language Benchmarks Game, ~2y ago



Can we build a language-independent VM framework in which many languages can be implemented (more) easily?

What is needed to generate code for a user program?

1. The user program

2. Expressed semantics of each language element

Combine the semantics of each element of the user program and generate code for the combined result.

This is what a traditional compiler does. But are there alternatives?

Compilation without a guest language compiler:
1. Metatracing

- Express the language semantics as a bytecode interpreter in a relatively high-level language

- Modify the interpreter to gather bytecode execution traces from the guest program

- Combine the traces with the interpreter's actions to generate code for each trace; like unrolling the interpreter.

- Together with some hints and optimizations, can generate pretty good code.

PyPy

- Originally, a Python VM written in a subset of Python, *RPython* (*R*estricted — types can be inferred, and it is easily translated). Generated C code or LLVM IR.
- Subsequently, a framework for the implementation of multiple languages via meta-tracing.
- *Tracing the meta-level: PyPy's tracing JIT compiler*, Bolz et al., 2009.
- Good performance for a variety of languages: Python, Ruby, Prolog, PHP, …

Compilation without a guest language compiler:

2. Partial evaluation of ASTs

- Express the language semantics as an AST interpreter in a relatively high-level language
- Combine the guest application's ASTs with the interpreter semantics; generate code

Example

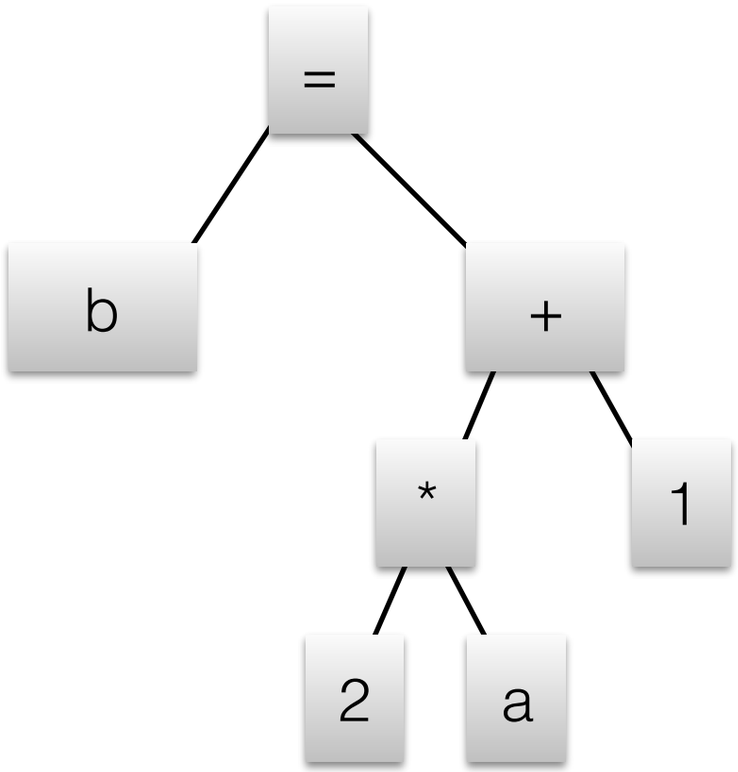
Consider a simple expression AST interpreter:

```
int eval(Exp *e) {
switch (e->tag) {
case CONST: return e->u.val;
case VAR: return vars[e->u.var];
case ADD: return eval(e->u.exp.l)+eval(e->u.exp.r);
/* ditto SUB, MUL and DIV */
case ASSGN: return vars[e->u.assgn.var]= eval(e->u.assgn.rhs); }
```

How can we compile code for an expression such as b=2*a+1?

Compiling a simple expression

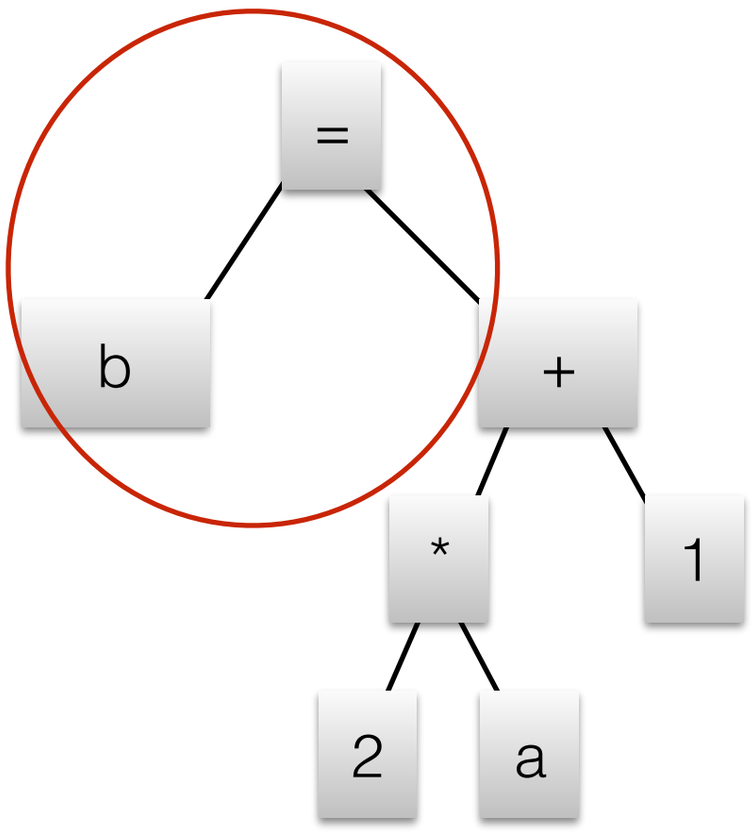
b=2*a+1




```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

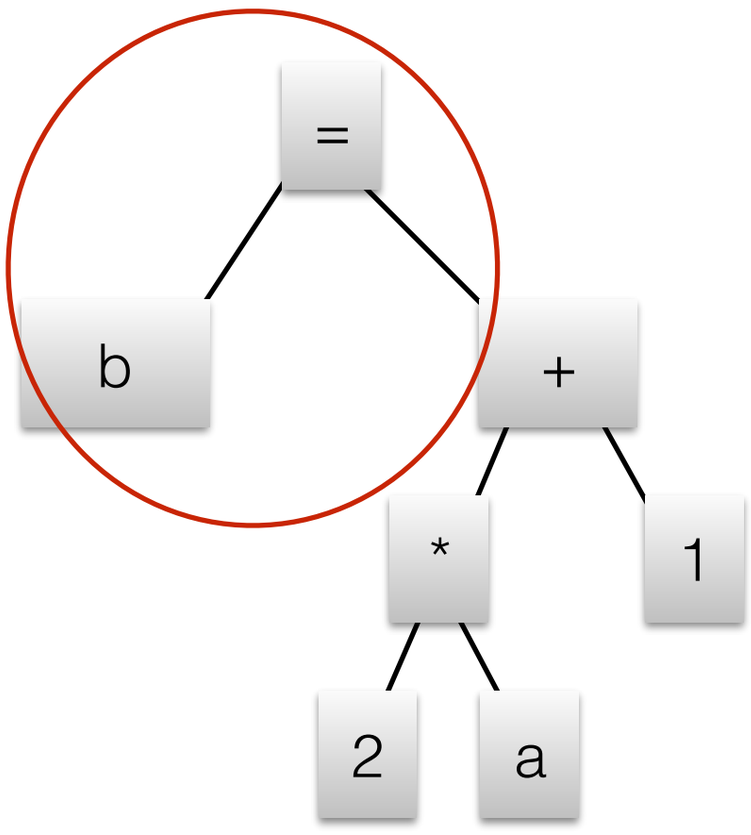
b=2*a+1




```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

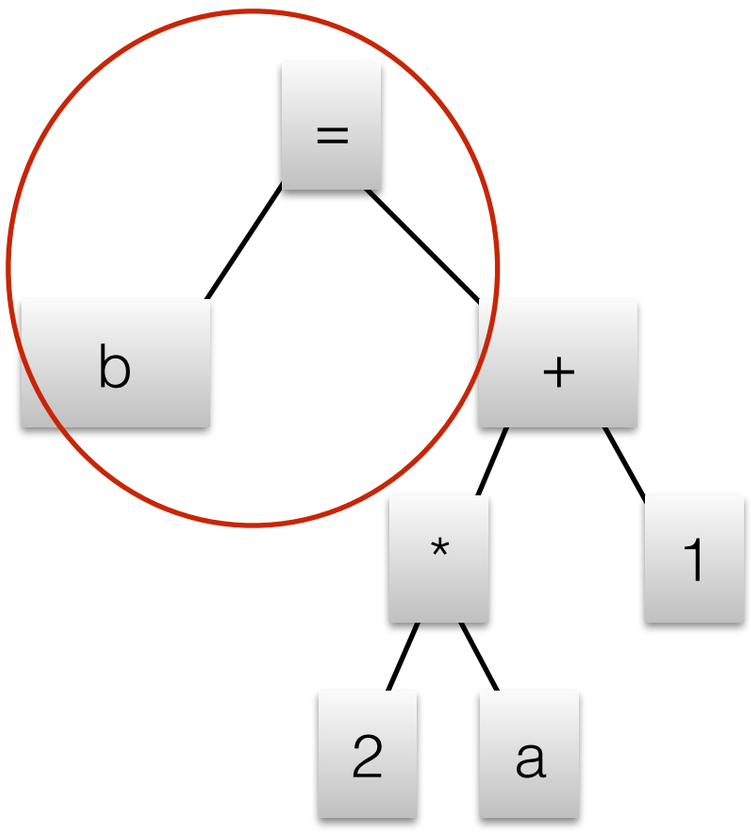
b=2*a+1




```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1



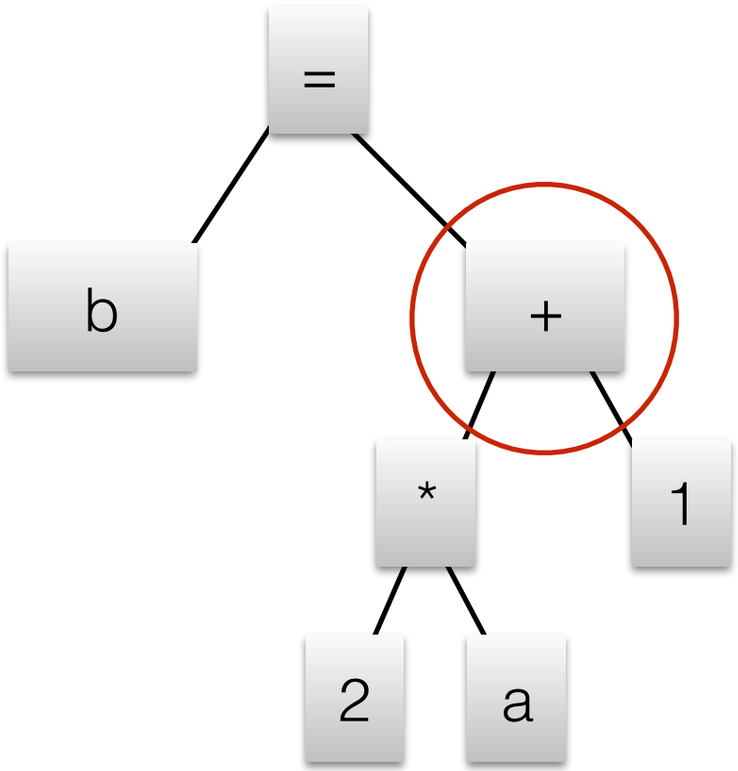
vars[‘b’]=

```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

vars[‘b’]=

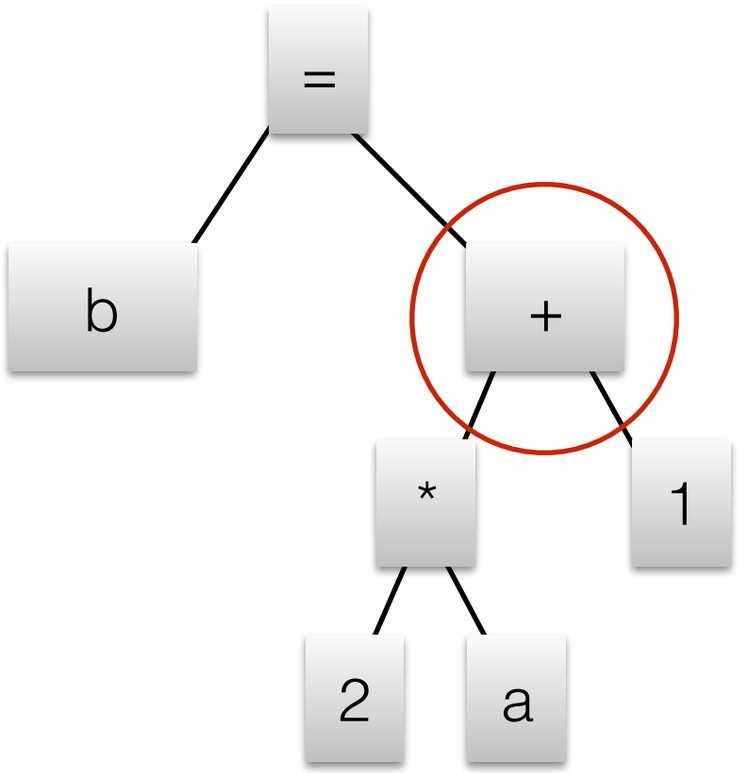



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                  +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

vars[‘b’]= ()+()

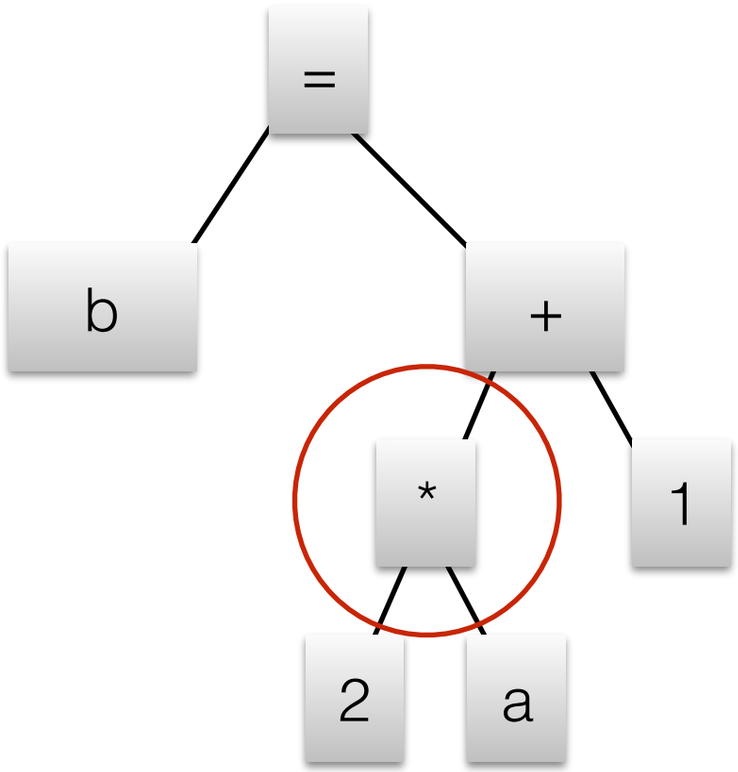



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                  +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

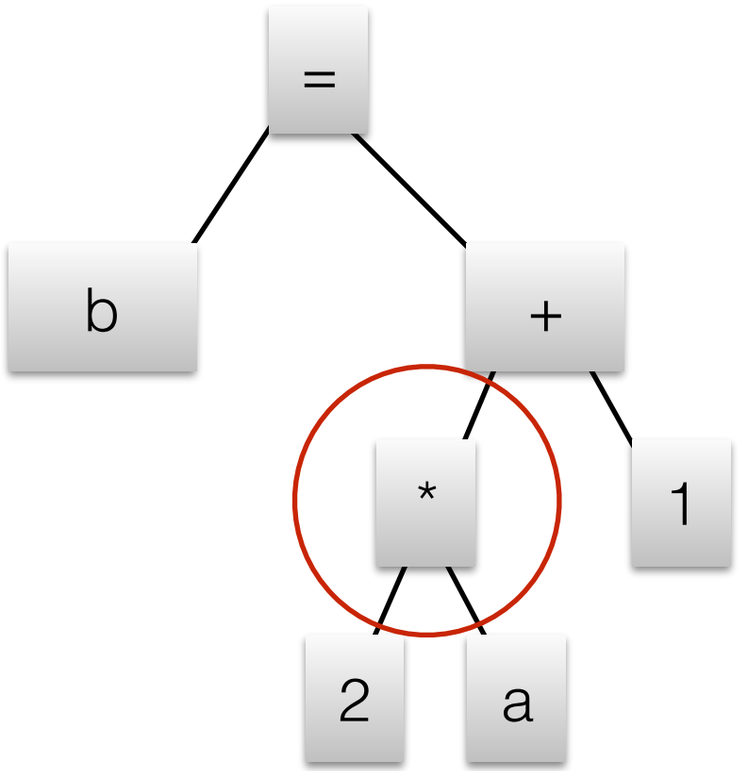
vars[‘b’]= ()+()




```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1



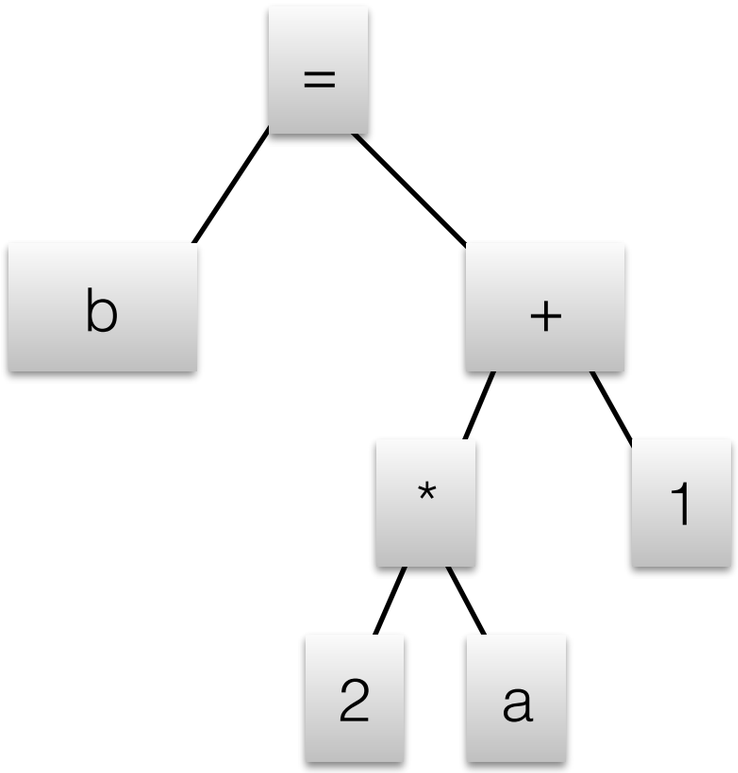

```
vars['b']= ()*()+()
```

```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

vars[‘b’]= ()*()+()

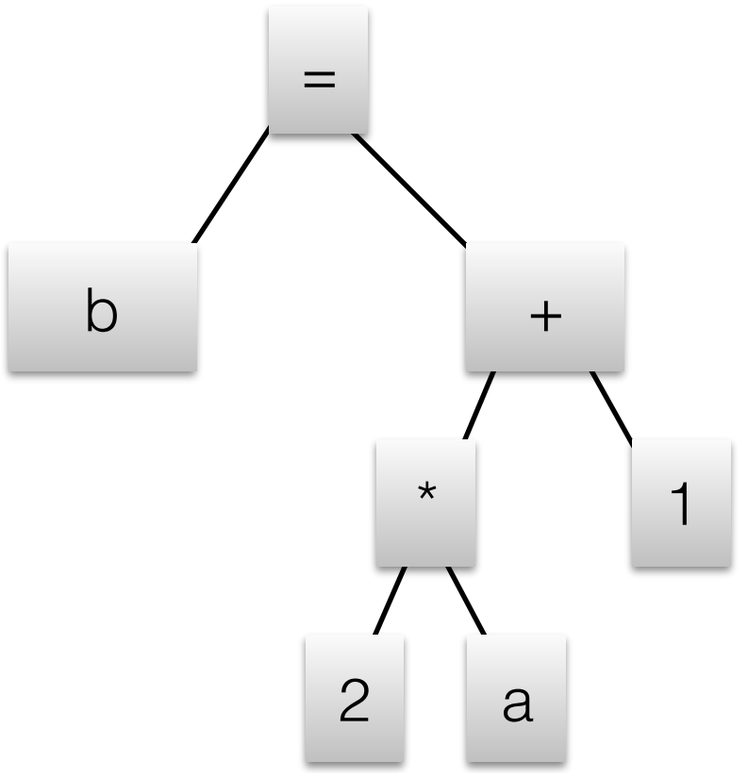



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

vars[‘b’]= 2 *()+()

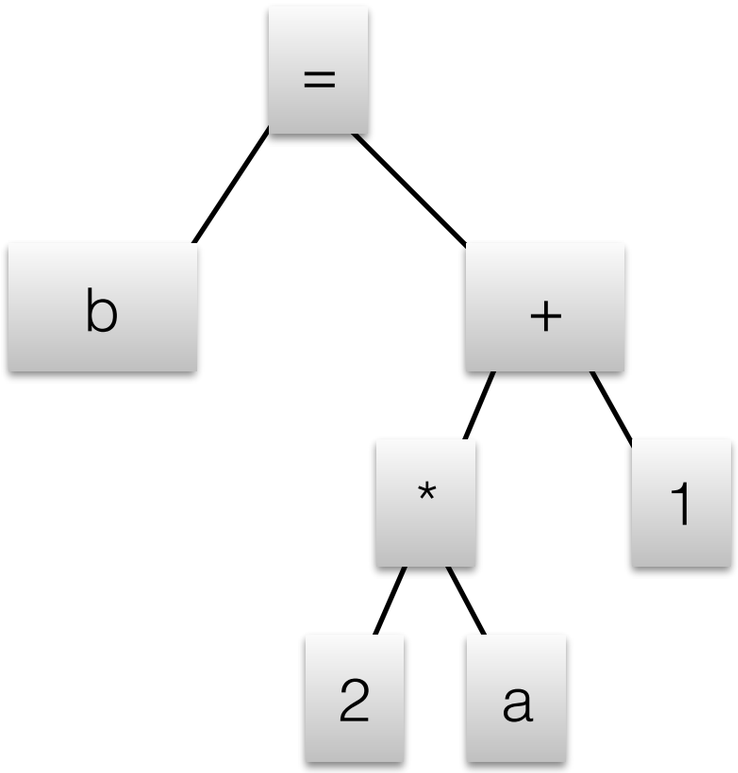



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

```
vars[‘b’]= 2 * vars[‘a’]+()
```

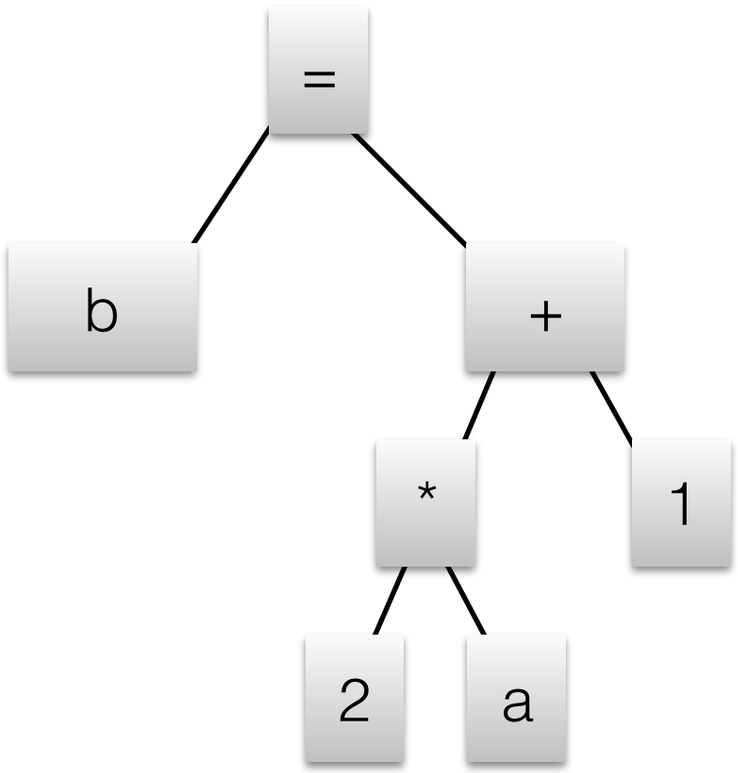



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

```
vars[‘b’]= 2 * vars[‘a’]+ 1
```

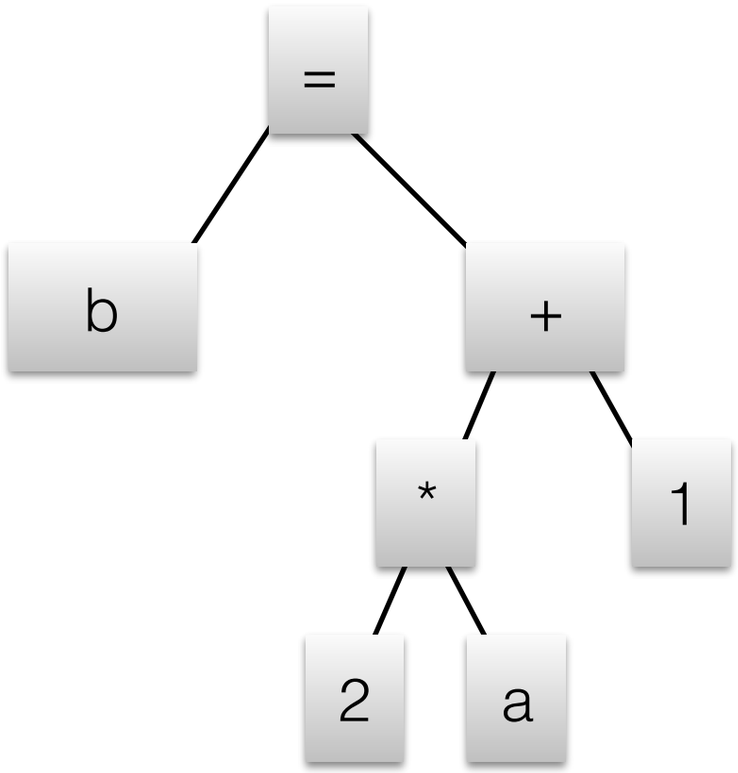



```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Compiling a simple expression

b=2*a+1

vars[‘b’]=




```
int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)
                   +eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return
    vars[e->u.assgn.var]= eval(e->u.assgn.rhs);
}
```

Partial evaluating the interpreter is compiling

- So one way to achieve language-independent compilation is to write a language interpreter and a partial evaluator for the language in which the *interpreter* is written

- To compile a different language, we just need a new interpreter, but not a new partial evaluator.

Partial evaluation alone is not enough

- Partial evaluation in this way has been known about for a long time [Futamura 71], but it hasn't helped in implementing dynamic languages efficiently. Why?

- The problem is the lack of type and other behavioral information, which only becomes manifest at run time.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element

- Express what the user program is doing/likely to do in concrete terms (hotspots, types)

- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
  - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)

- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
  - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
  - ✓Profile and specialize within the AST
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
  - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
  - ✓Profile and specialize within the AST
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.
  - ✓Use the interpreter and the profiles to generate specialized code

Specializing ASTs during interpretation

- One solution is to gather profile data during AST interpretation.

- But to get faster interpretation *and* profiling data, we can specialize the AST nodes at run-time based on each node's observed behavior.

Example: addition

```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

Example: addition

```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

Example: addition

```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

```
String add(String a,
           String b) {
   return a + b;
}
```

Example: addition

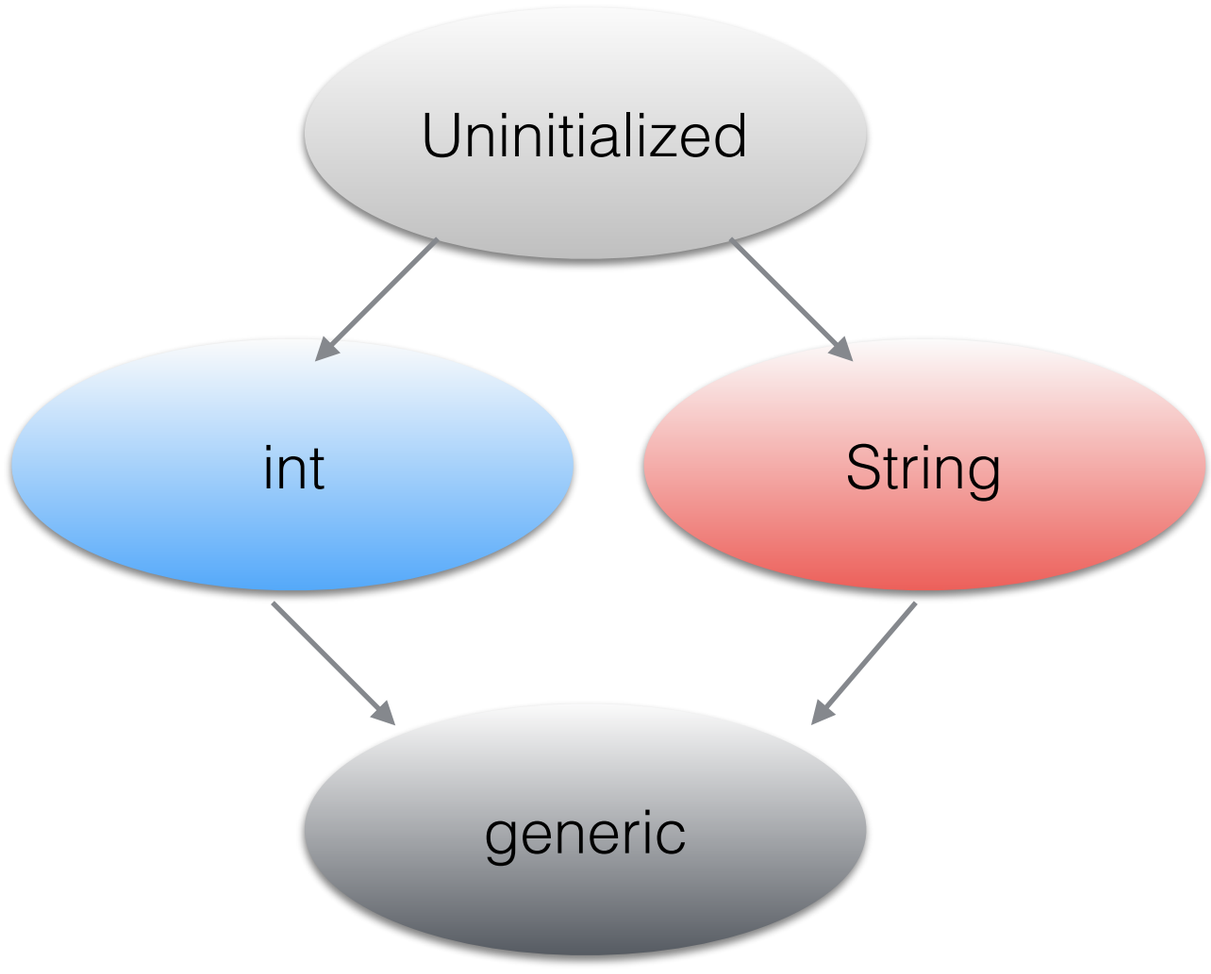
```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

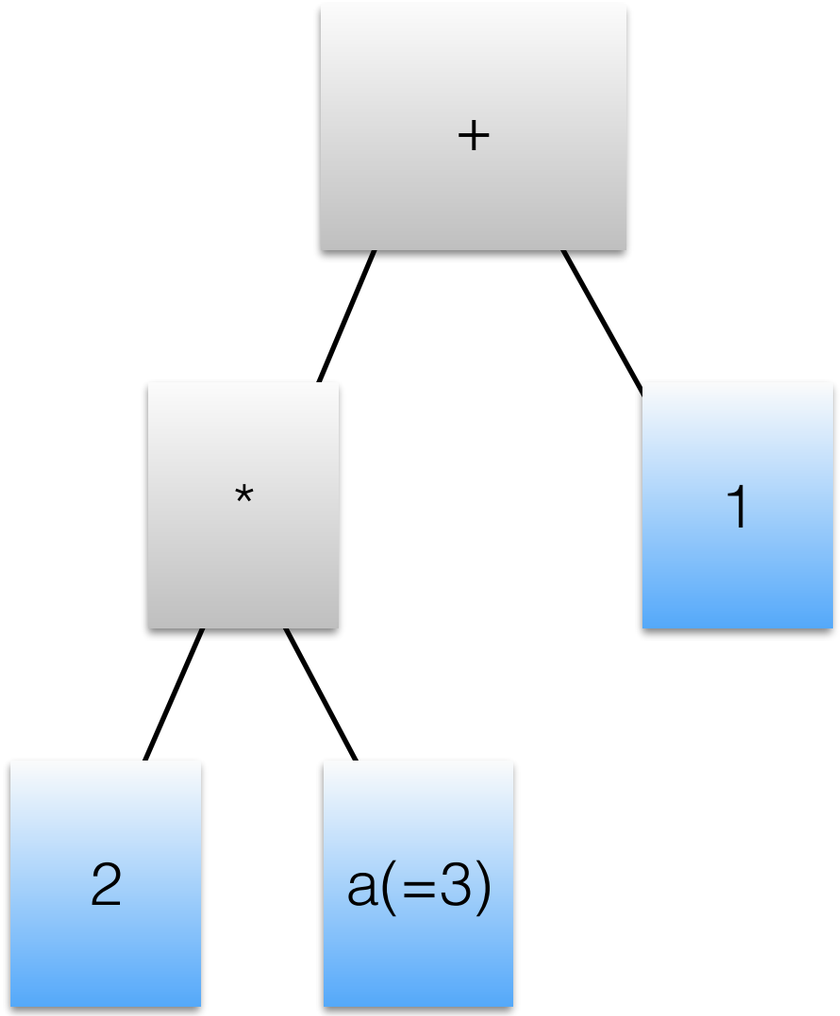
```
String add(String a,
           String b) {
   return a + b;
}
```

```
Object add(Object a,
           Object b) {
   return genericAdd(a, b);
}
```

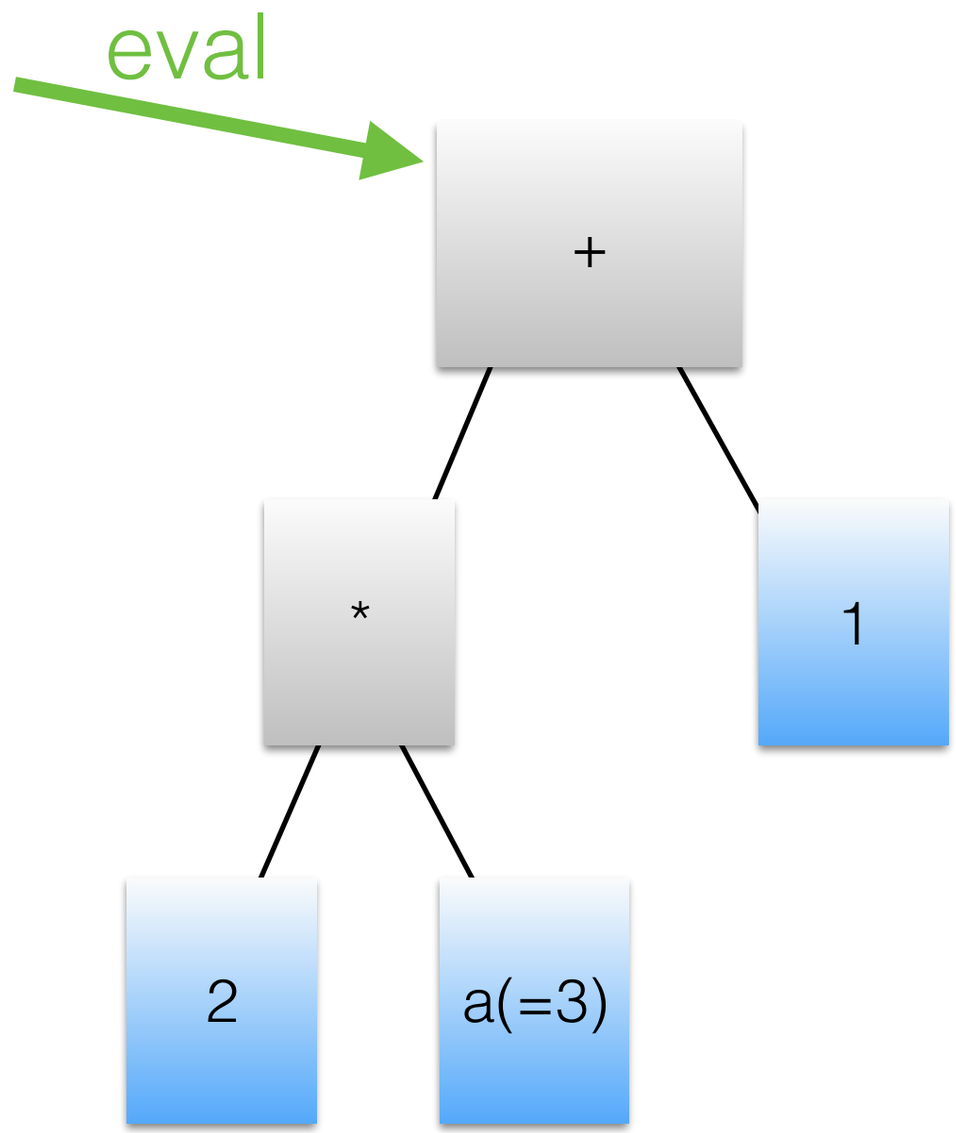
Type transitions



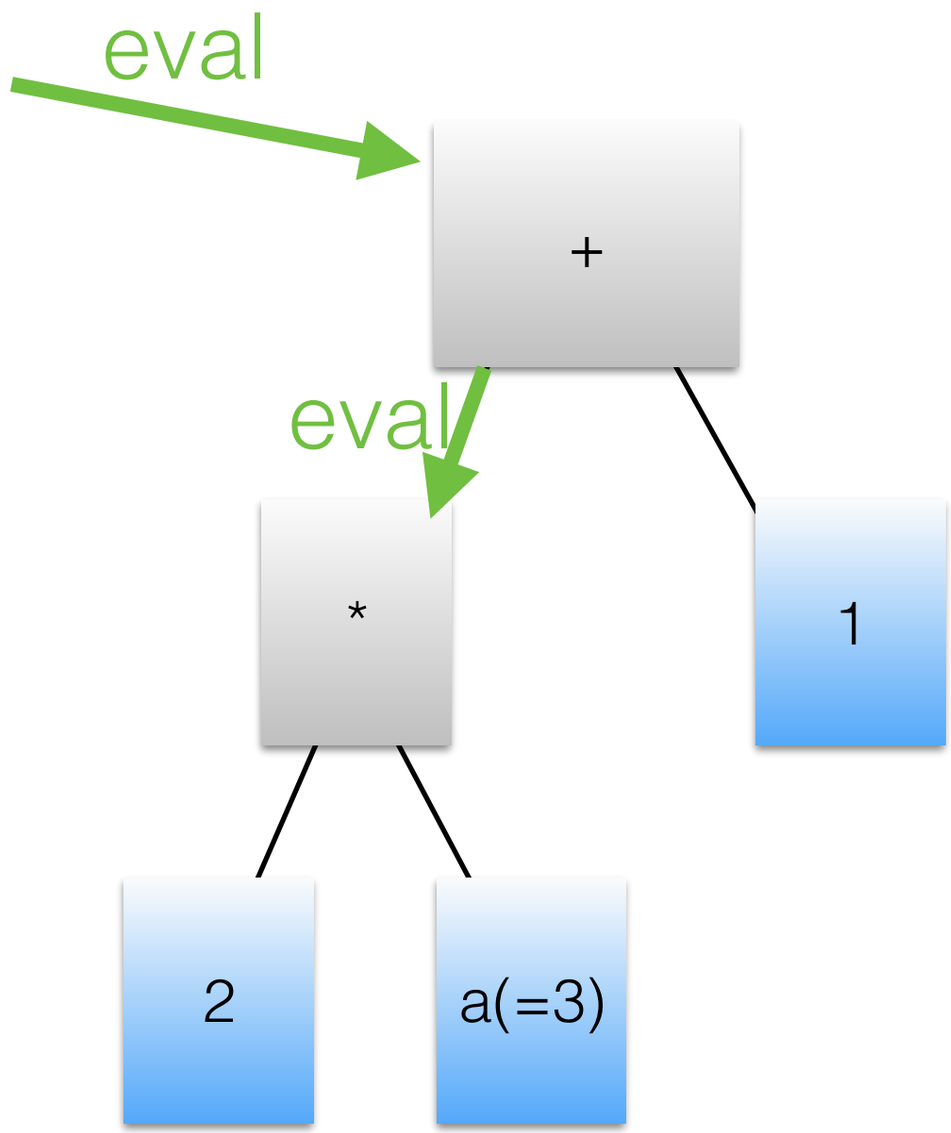
Evolution of an expression
1. Specialization



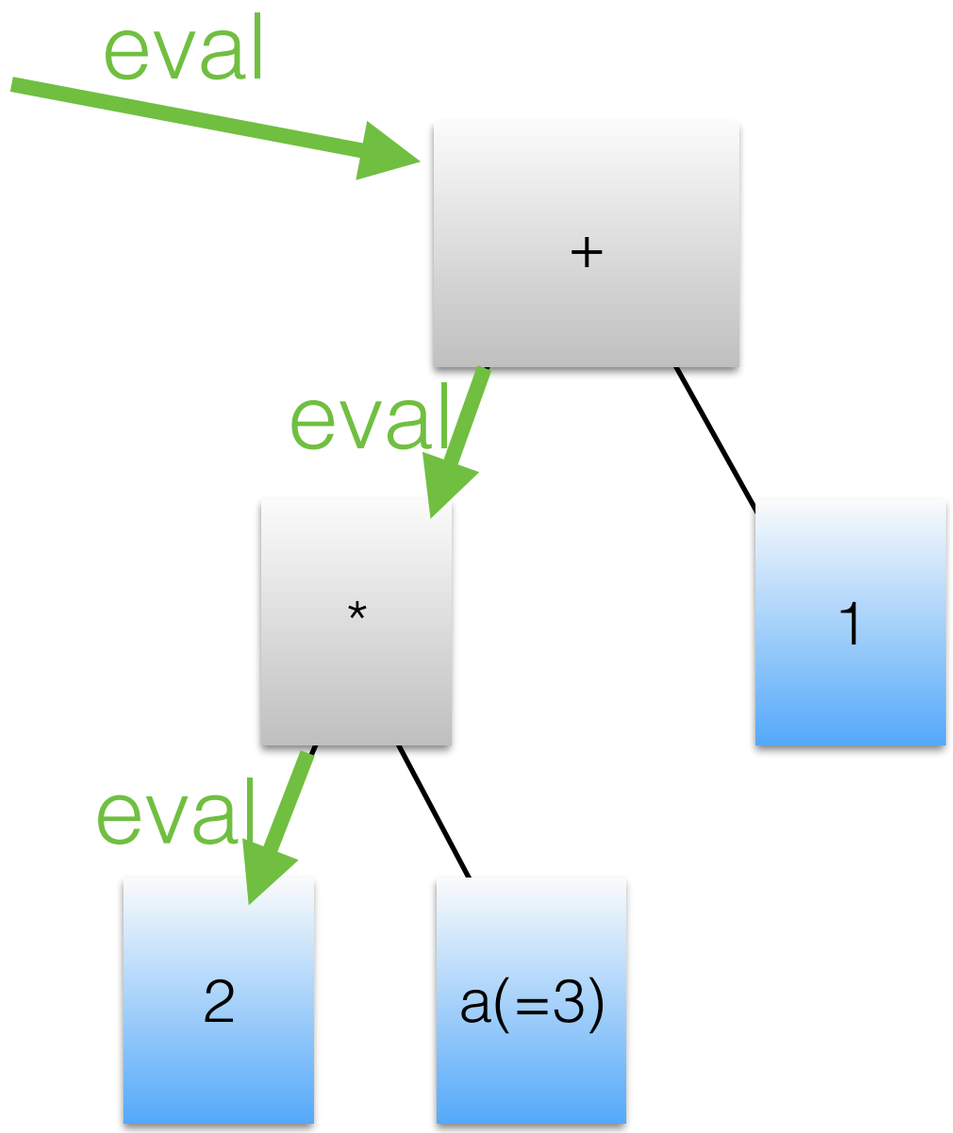
Evolution of an expression
1. Specialization



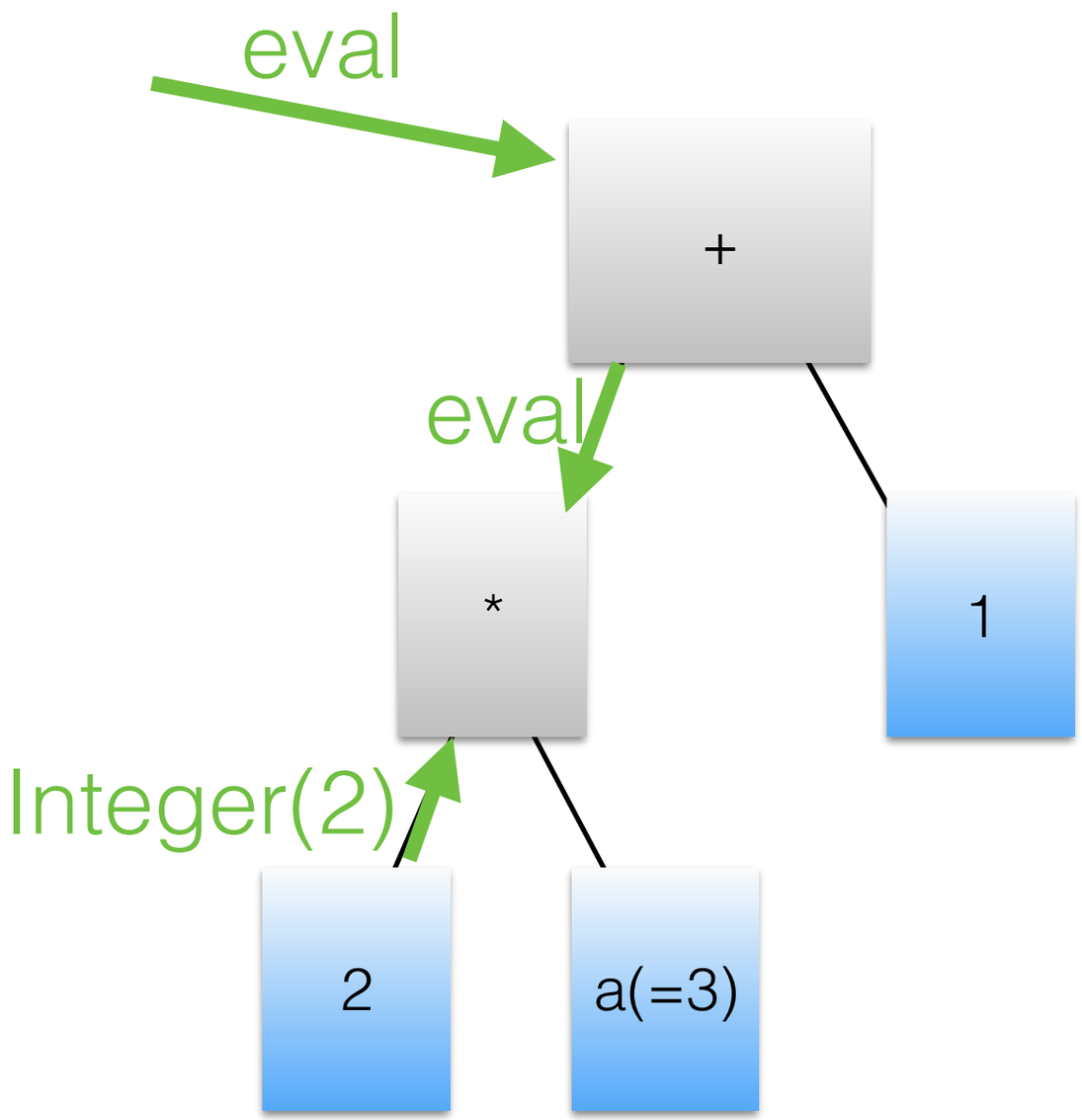
Evolution of an expression
1. Specialization



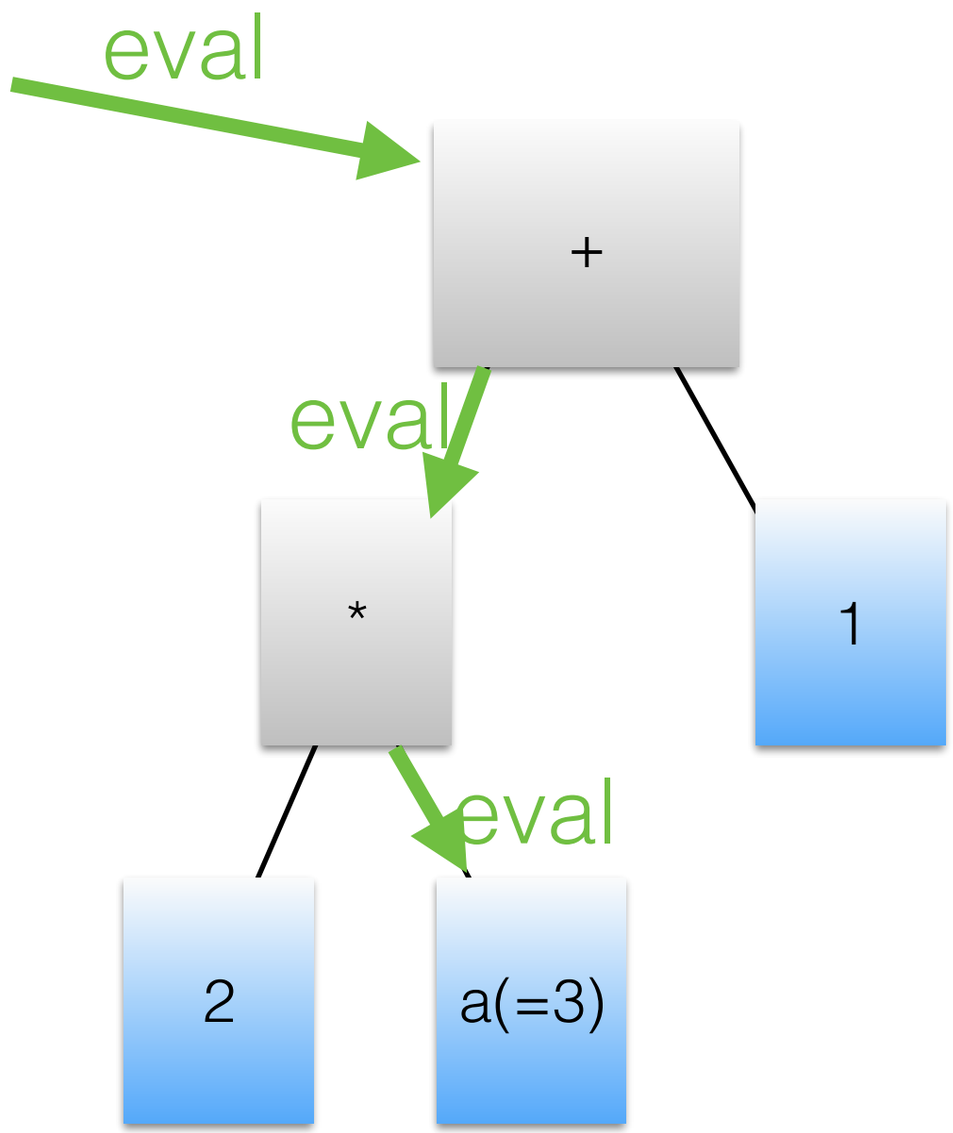
Evolution of an expression
1. Specialization



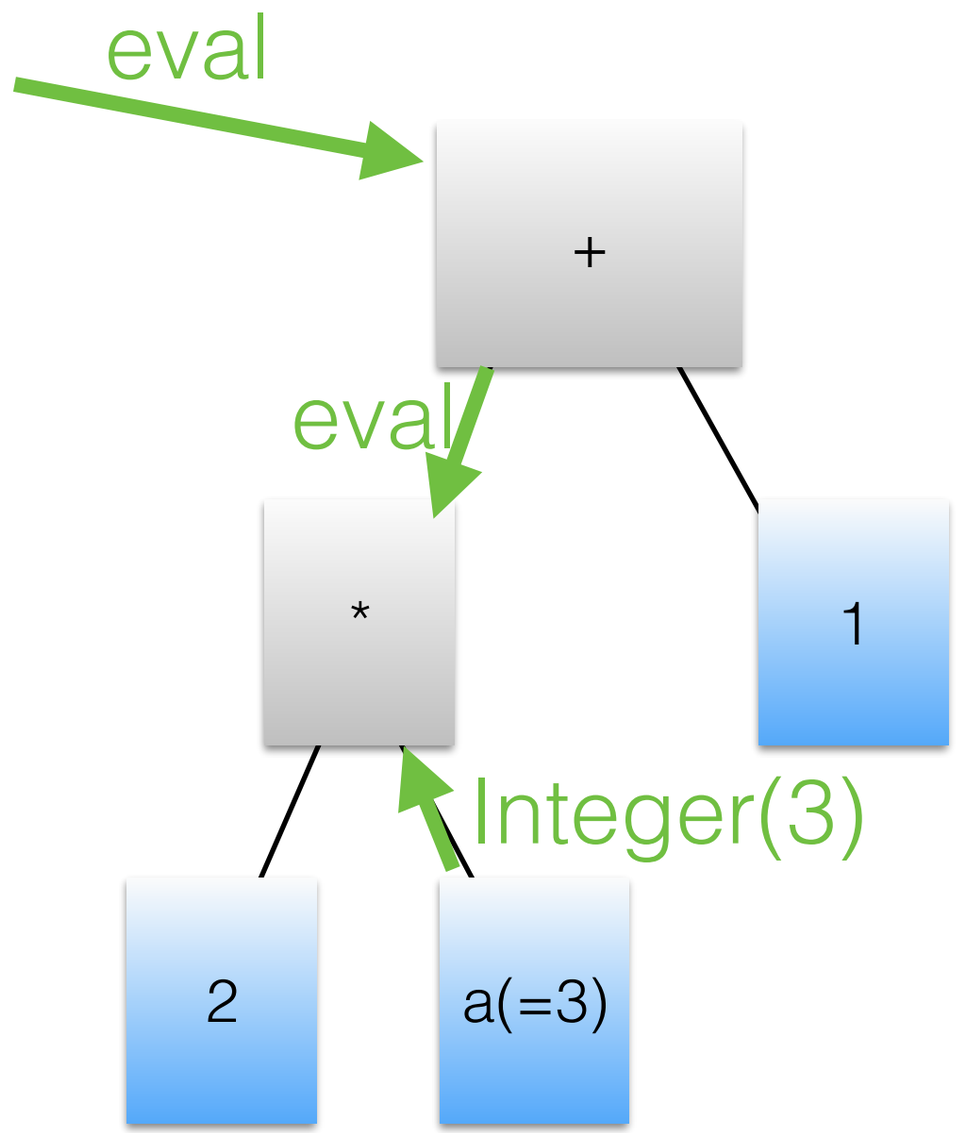
Evolution of an expression
1. Specialization



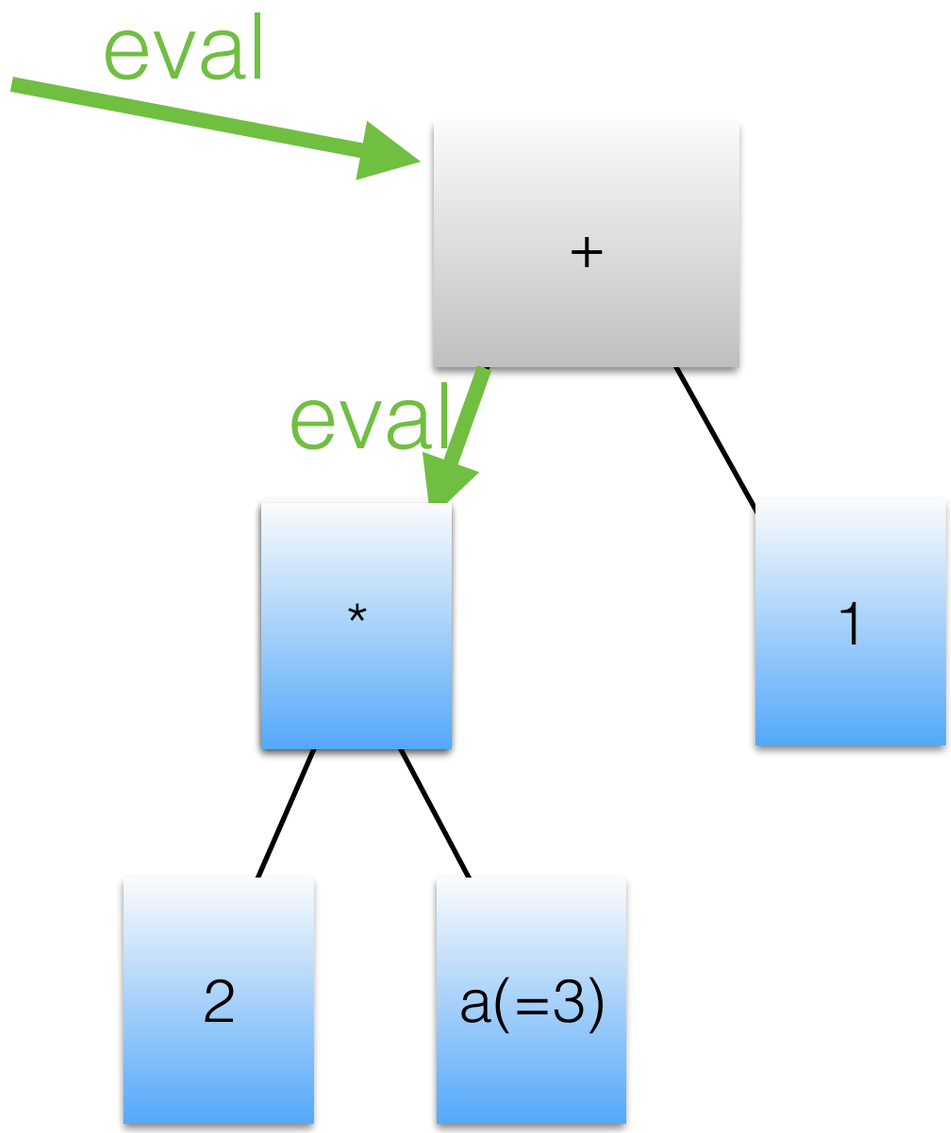
Evolution of an expression
1. Specialization



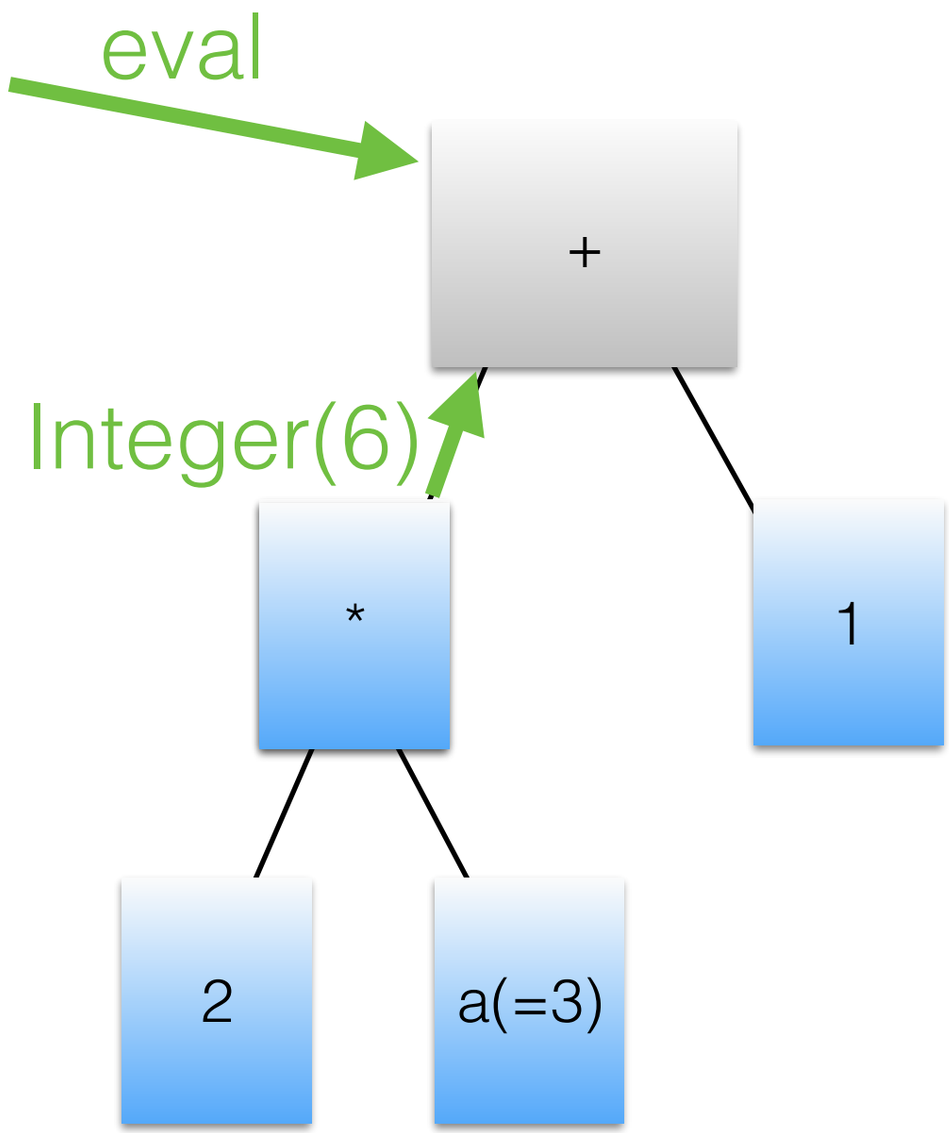
Evolution of an expression
1. Specialization



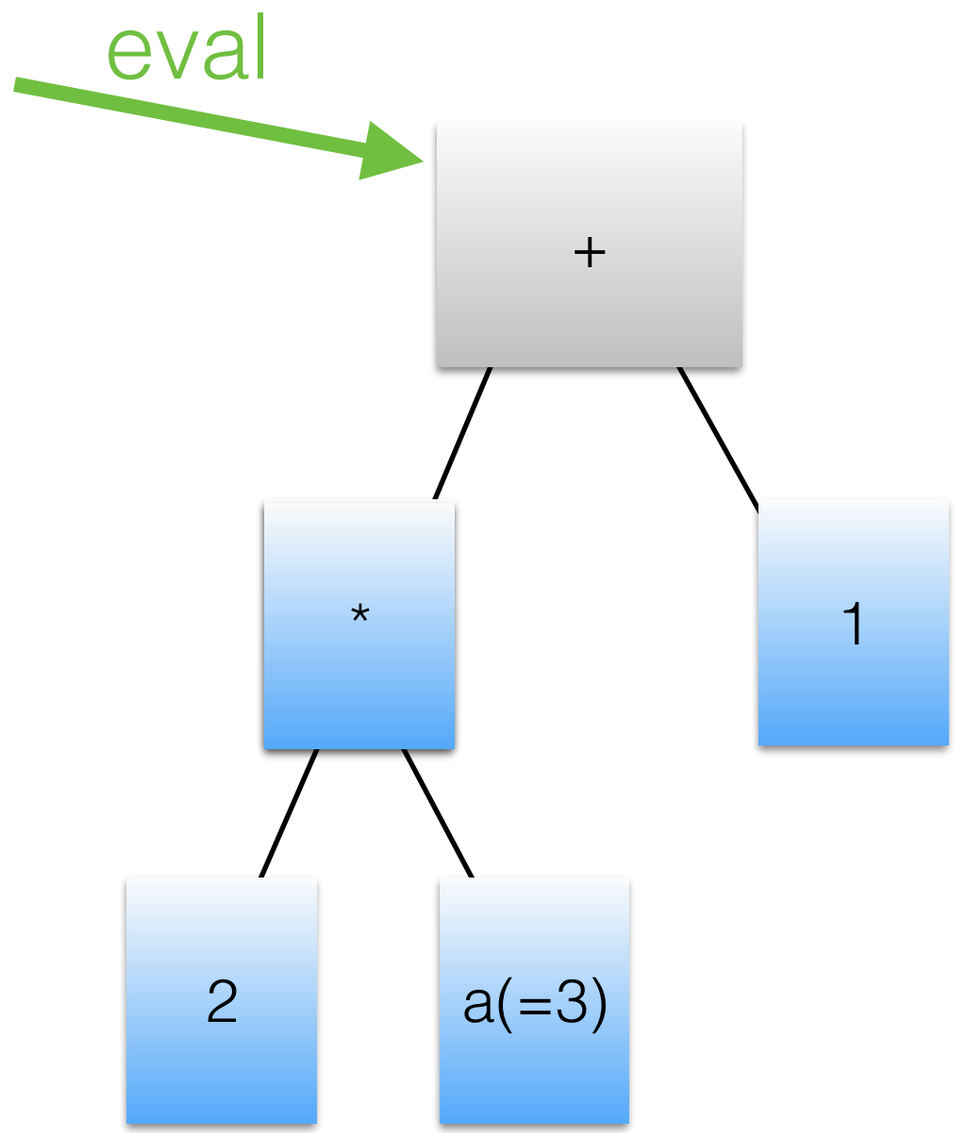
Evolution of an expression
1. Specialization



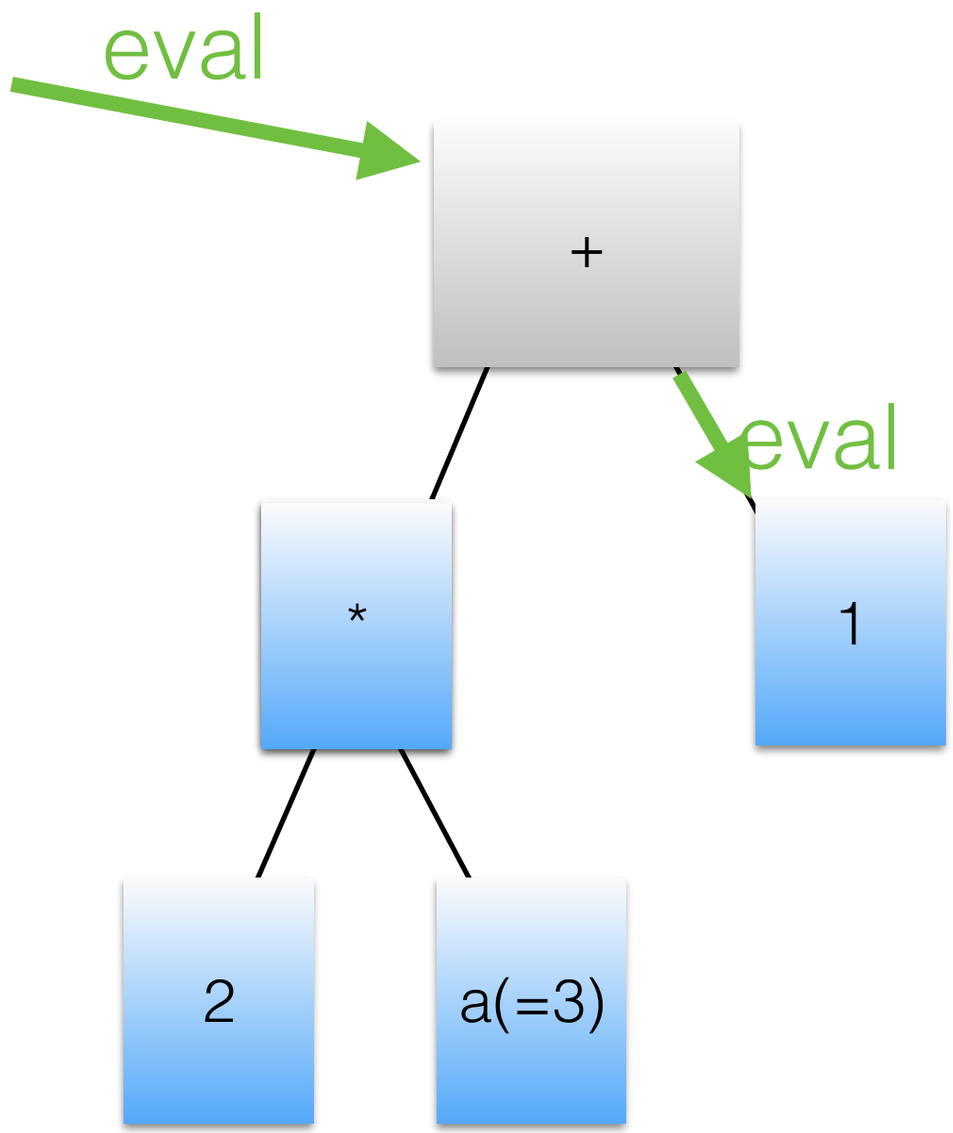
Evolution of an expression
1. Specialization



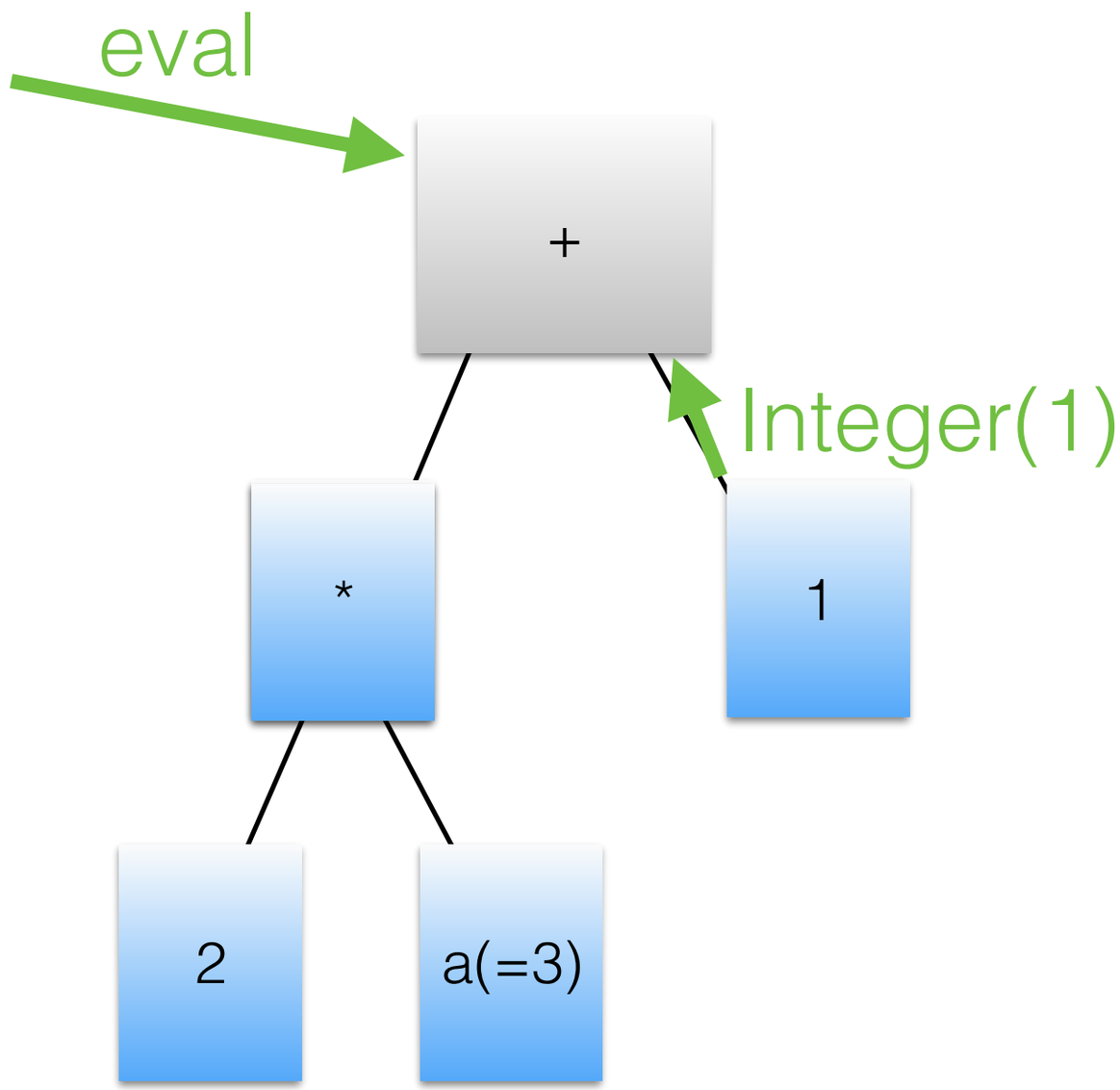
Evolution of an expression
1. Specialization



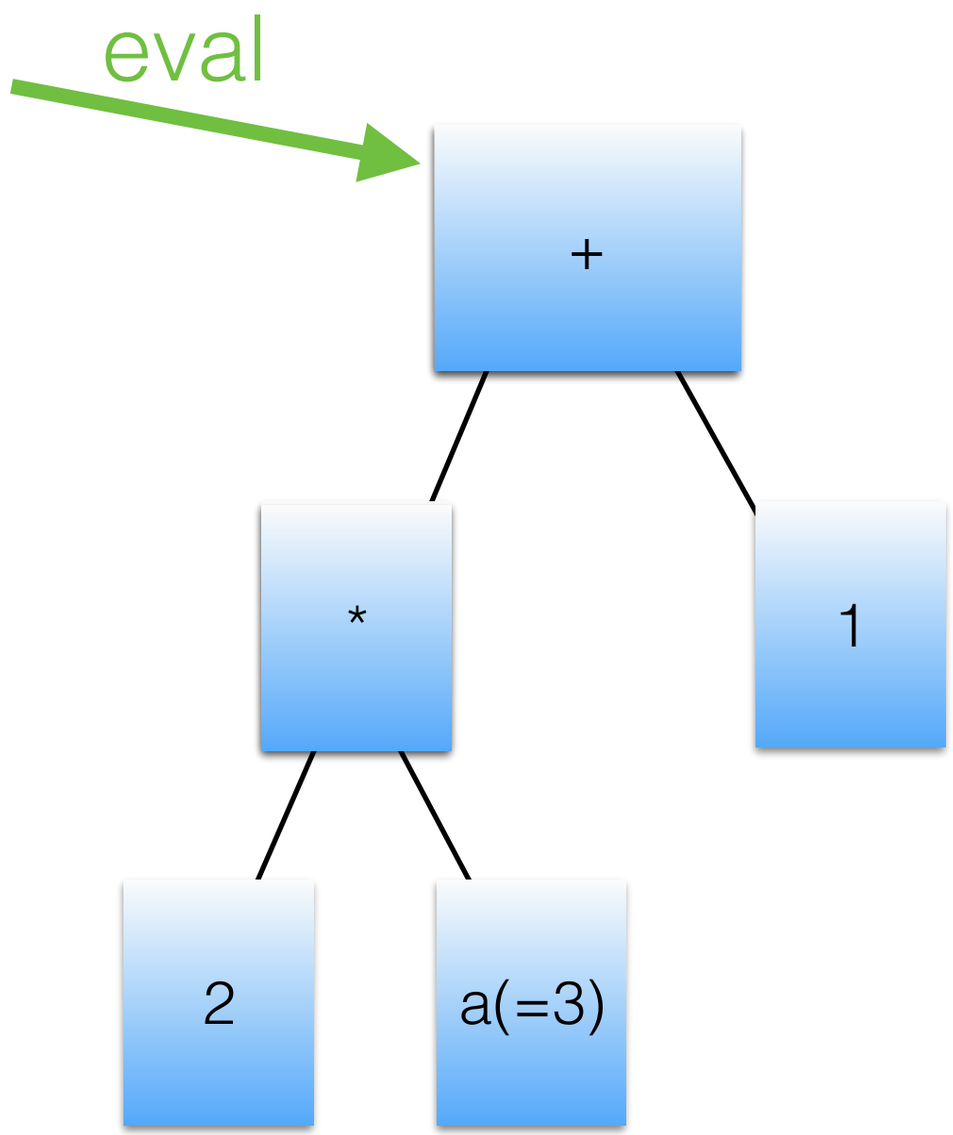
Evolution of an expression
1. Specialization



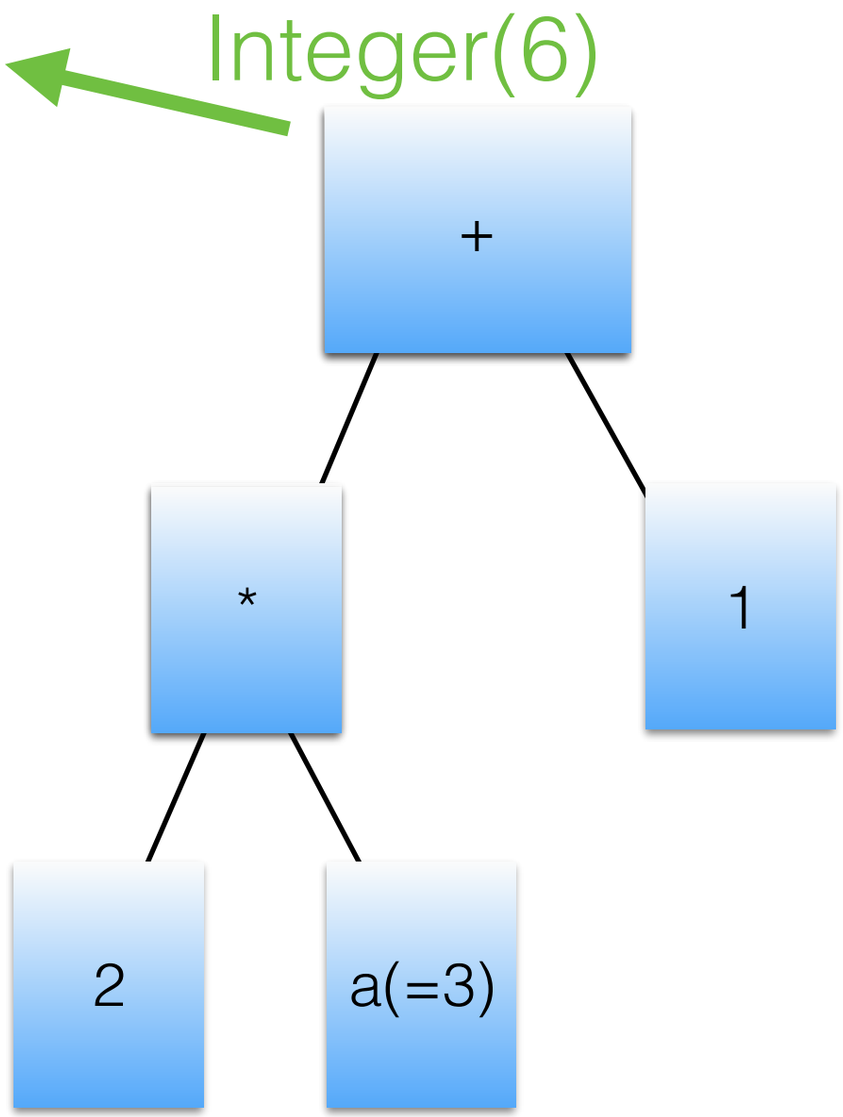
Evolution of an expression
1. Specialization



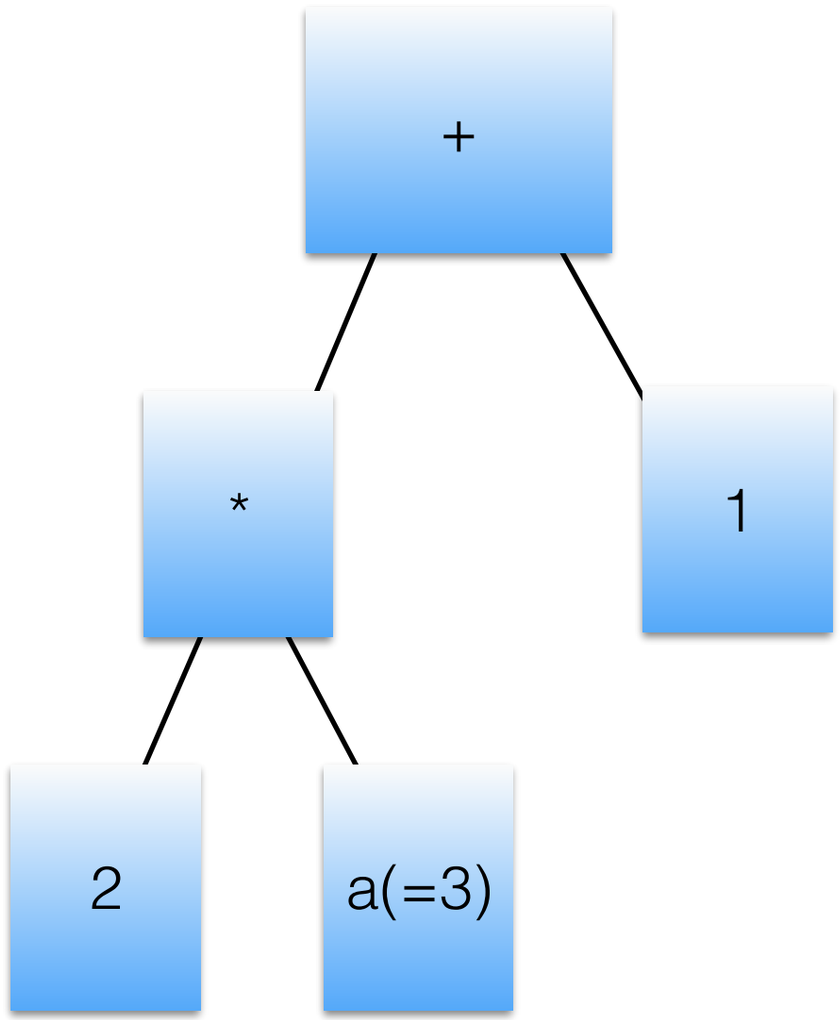
Evolution of an expression
1. Specialization



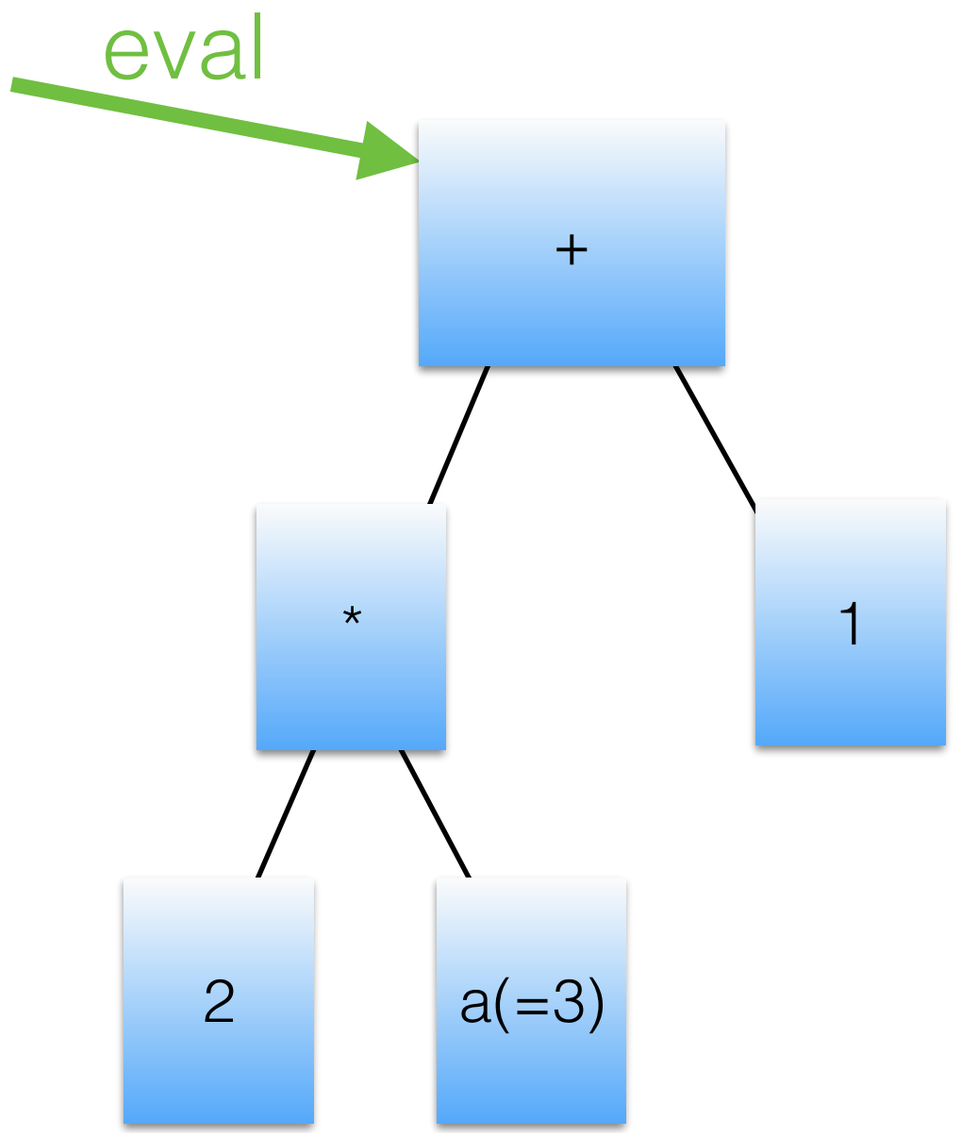
Evolution of an expression
1. Specialization



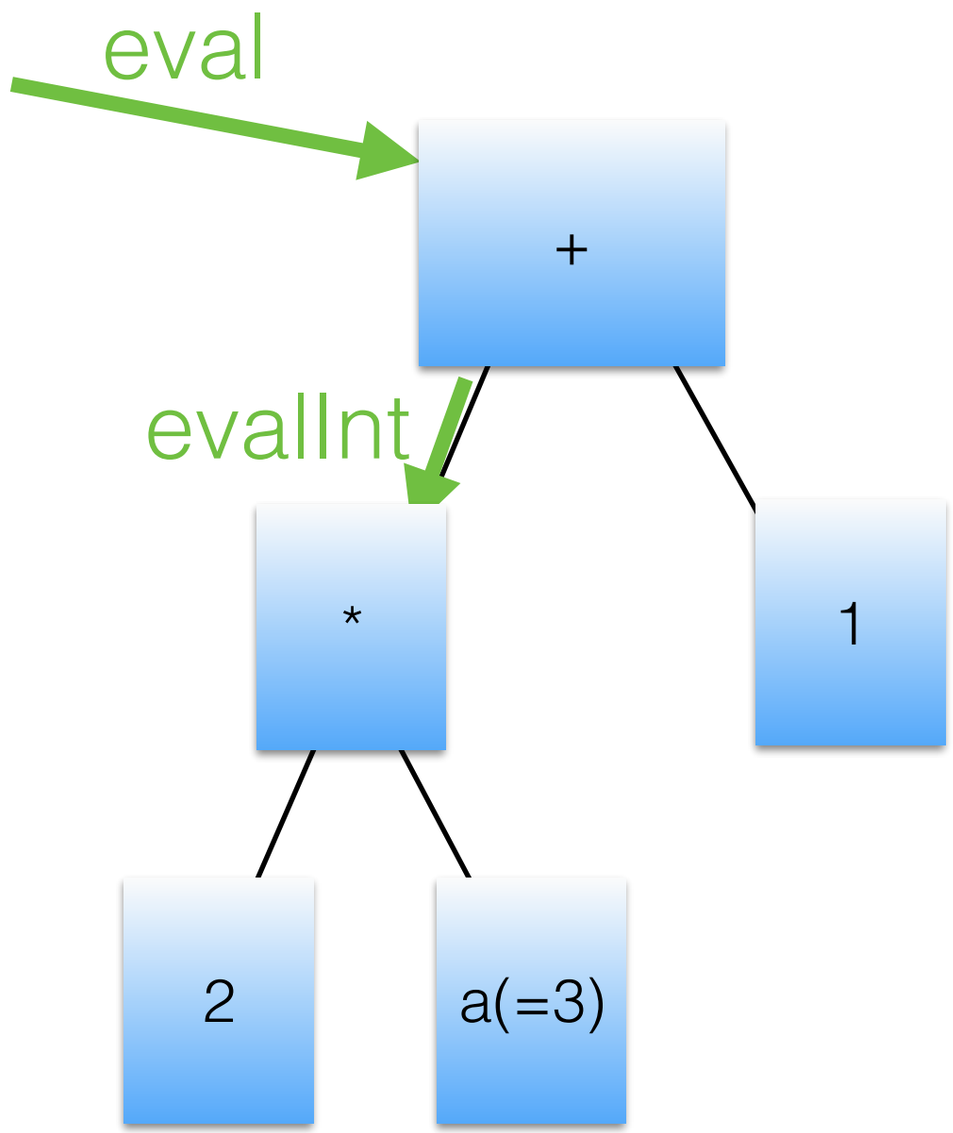
Evolution of an expression
2. Repeated execution



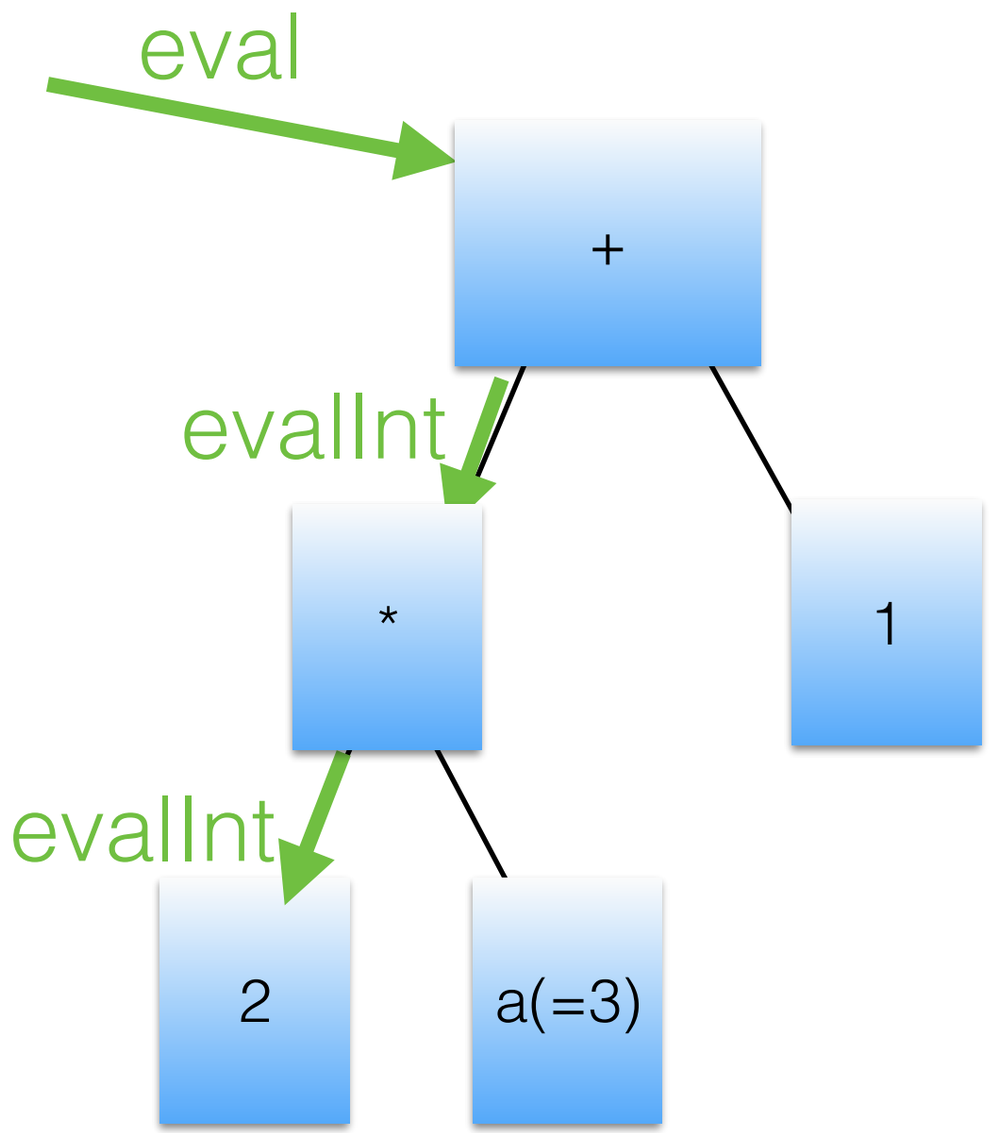
Evolution of an expression
2. Repeated execution



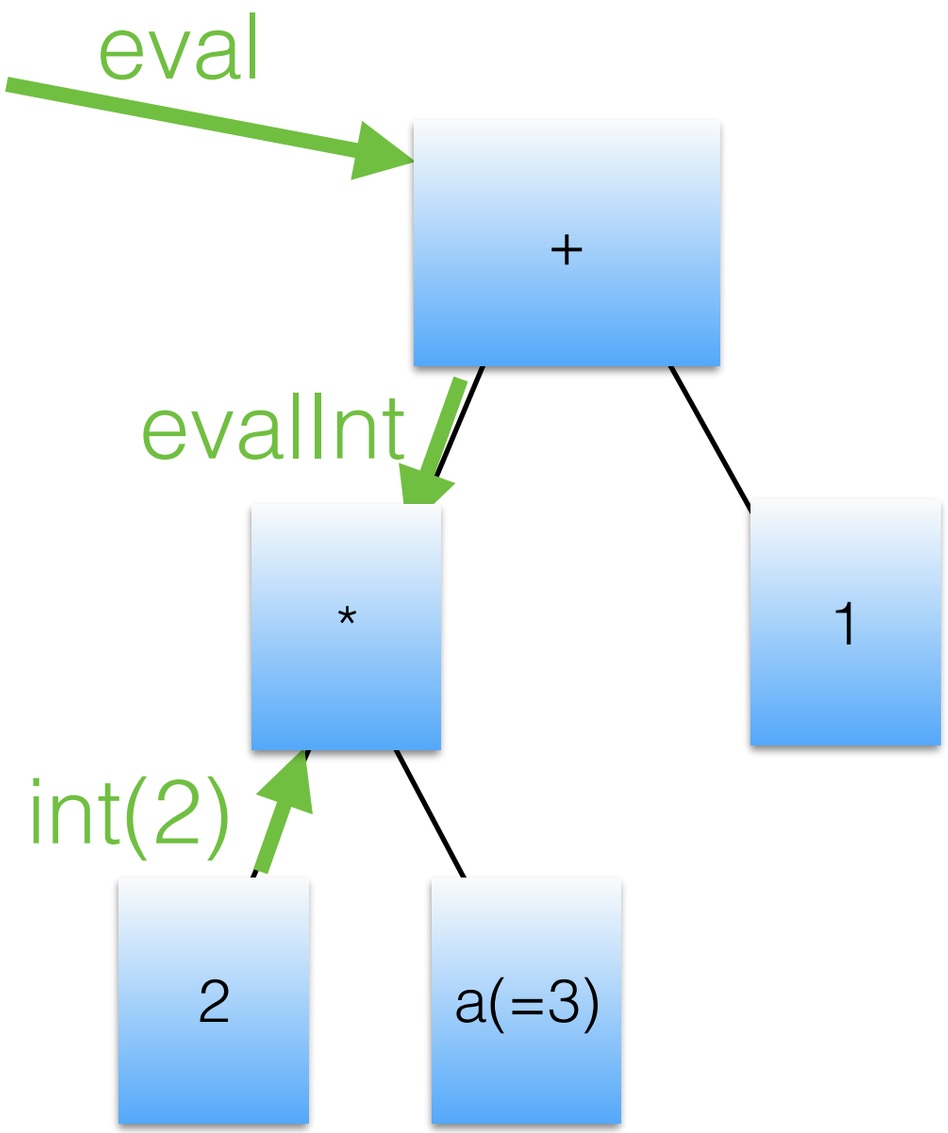
Evolution of an expression
2. Repeated execution



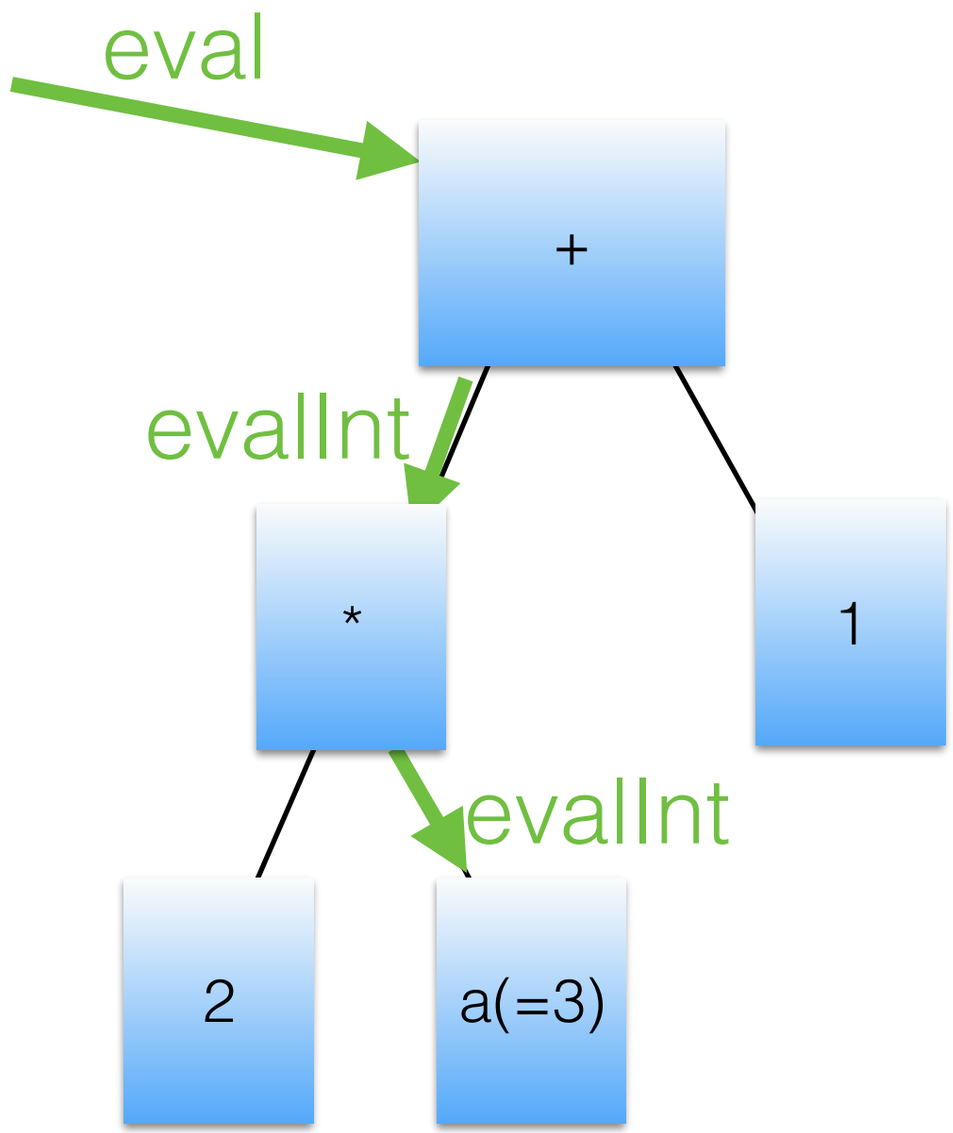
Evolution of an expression
2. Repeated execution



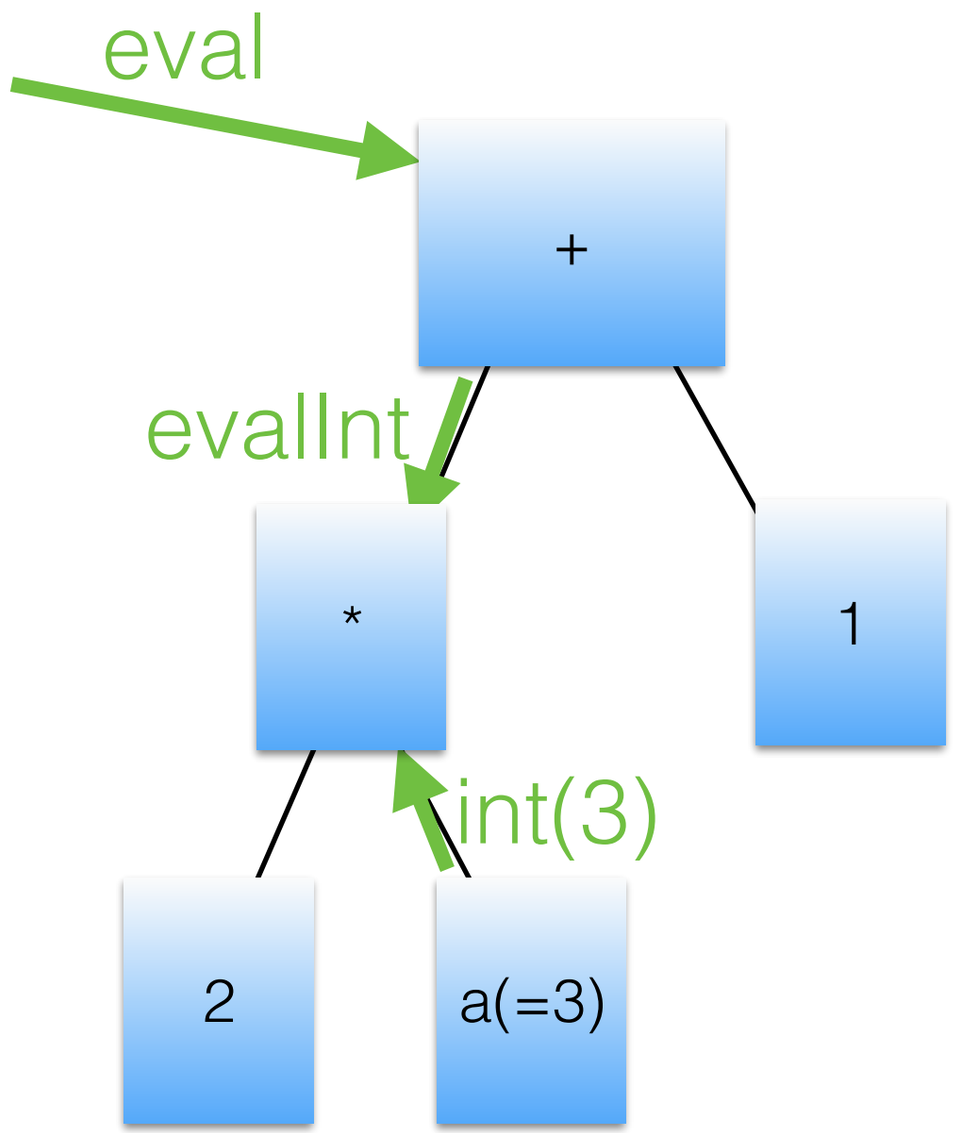
Evolution of an expression
2. Repeated execution



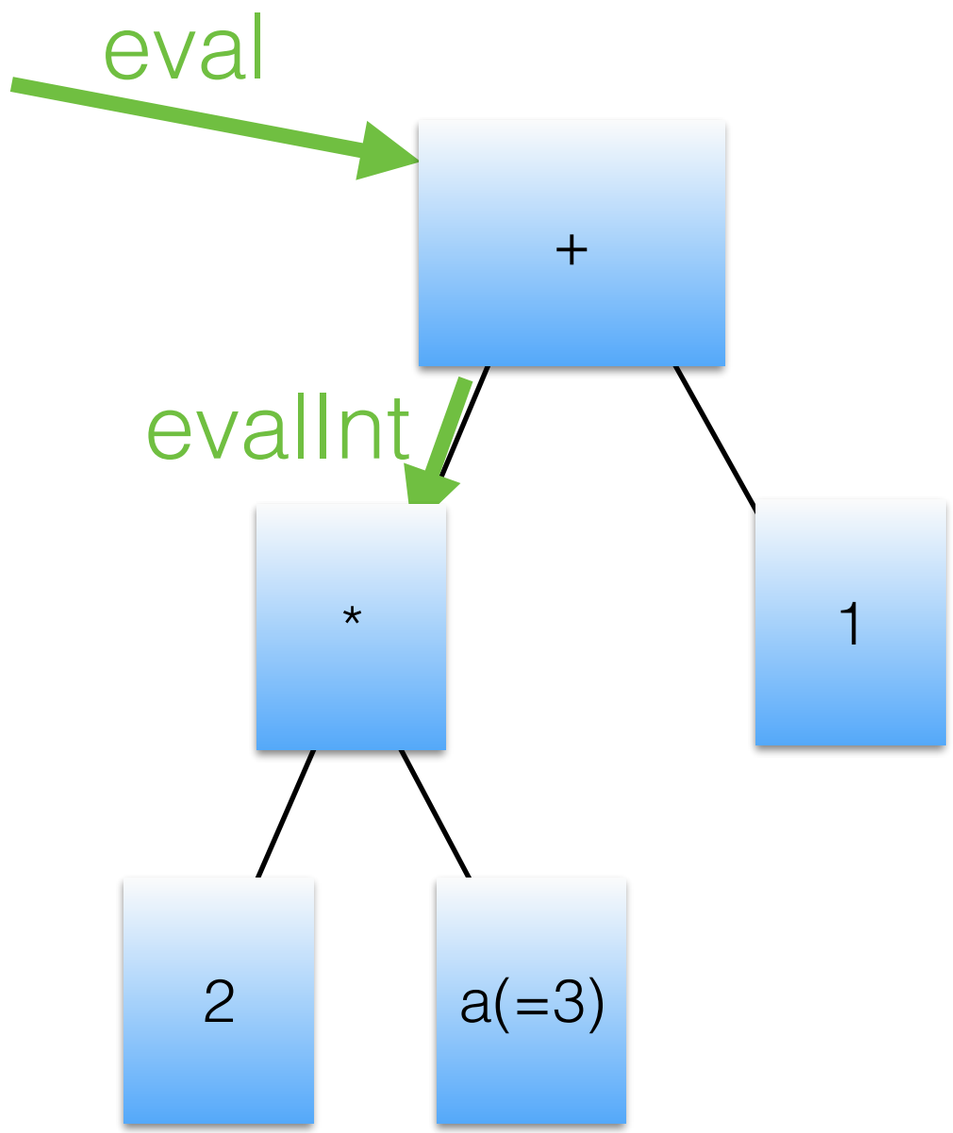
Evolution of an expression
2. Repeated execution



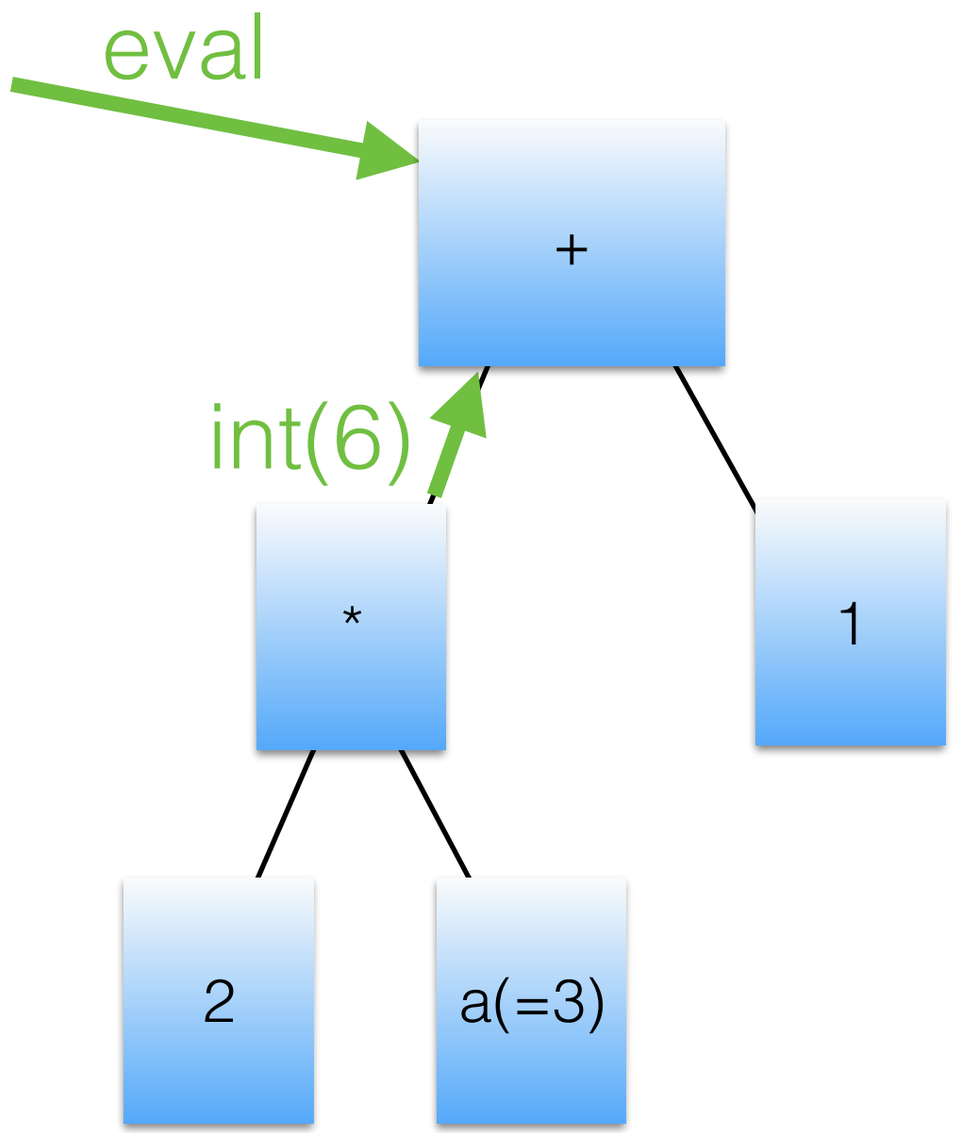
Evolution of an expression
2. Repeated execution



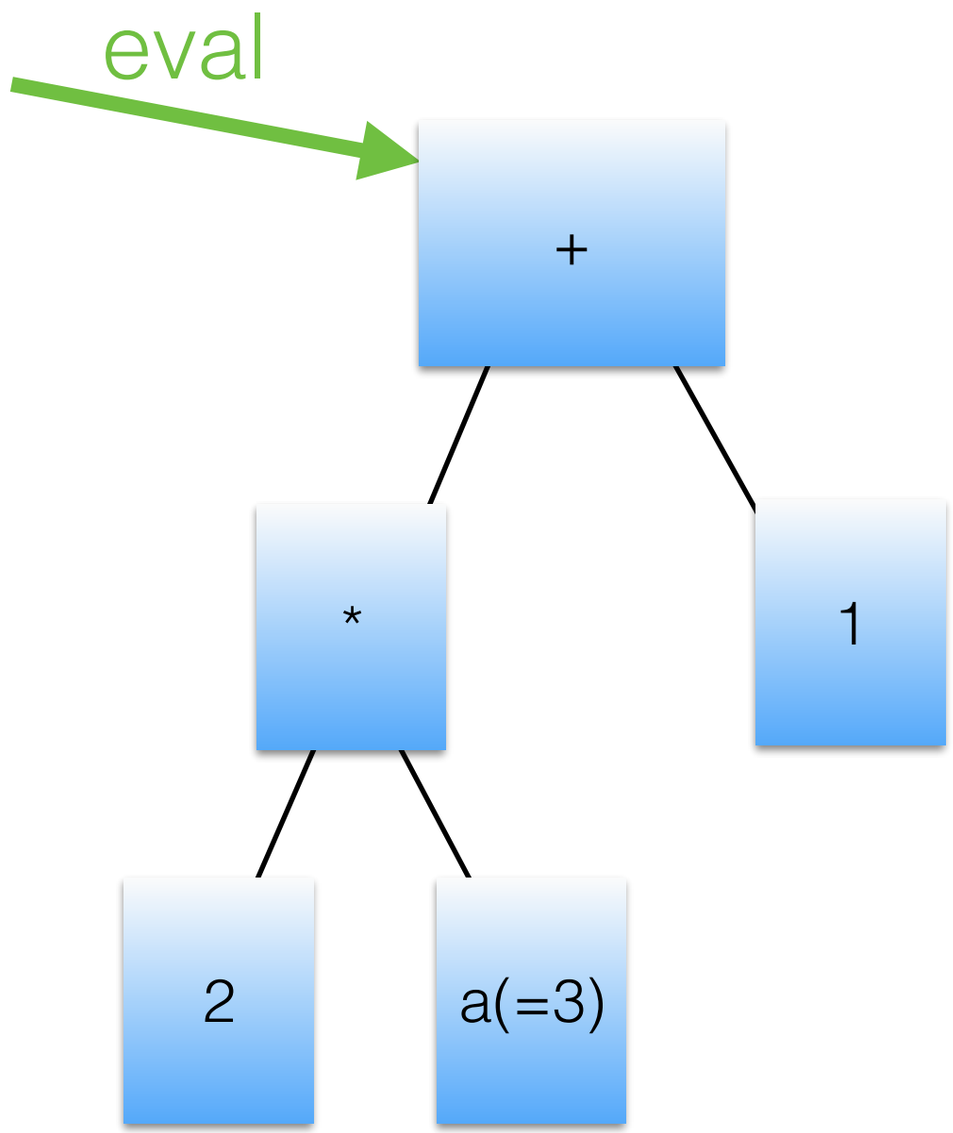
Evolution of an expression
2. Repeated execution



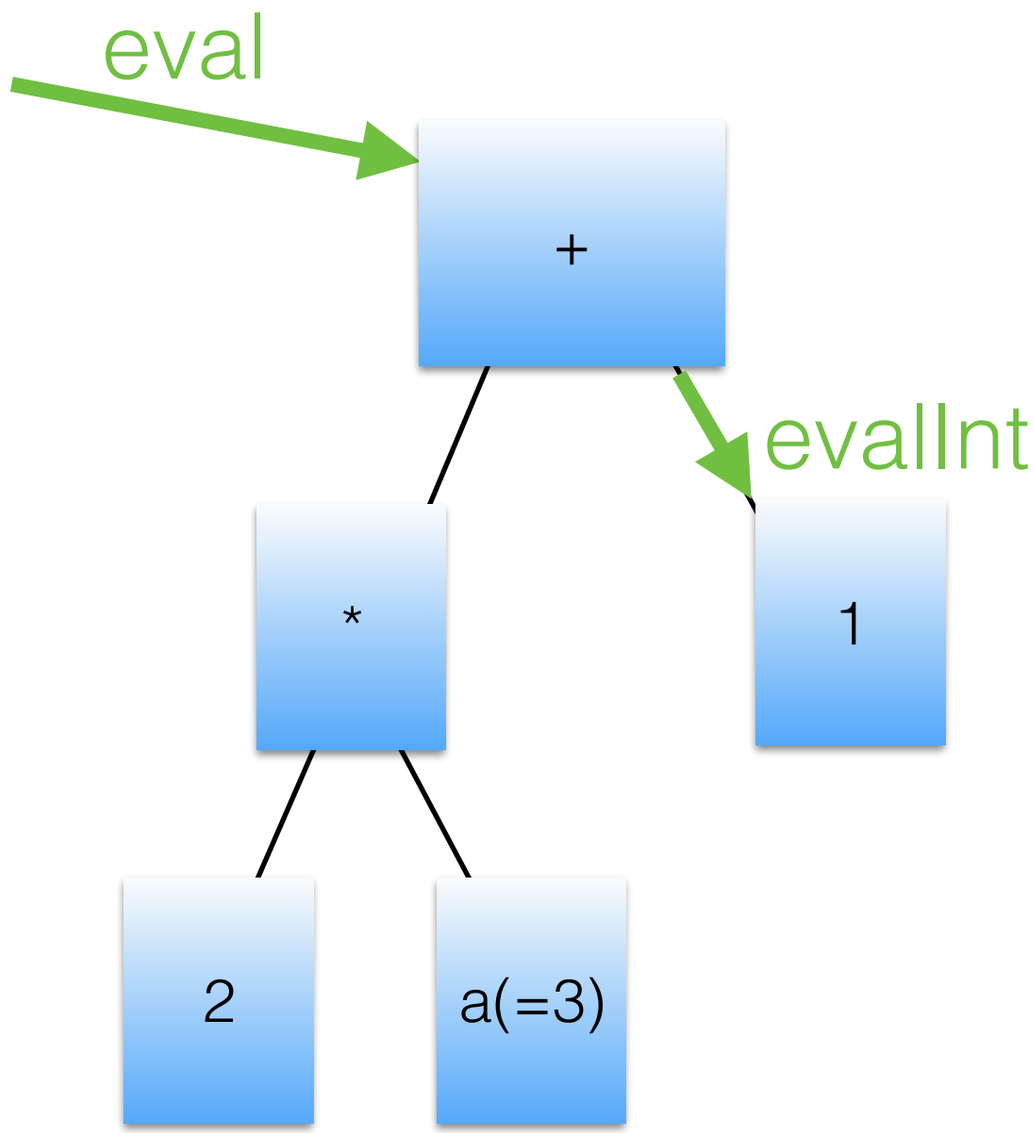
Evolution of an expression
2. Repeated execution



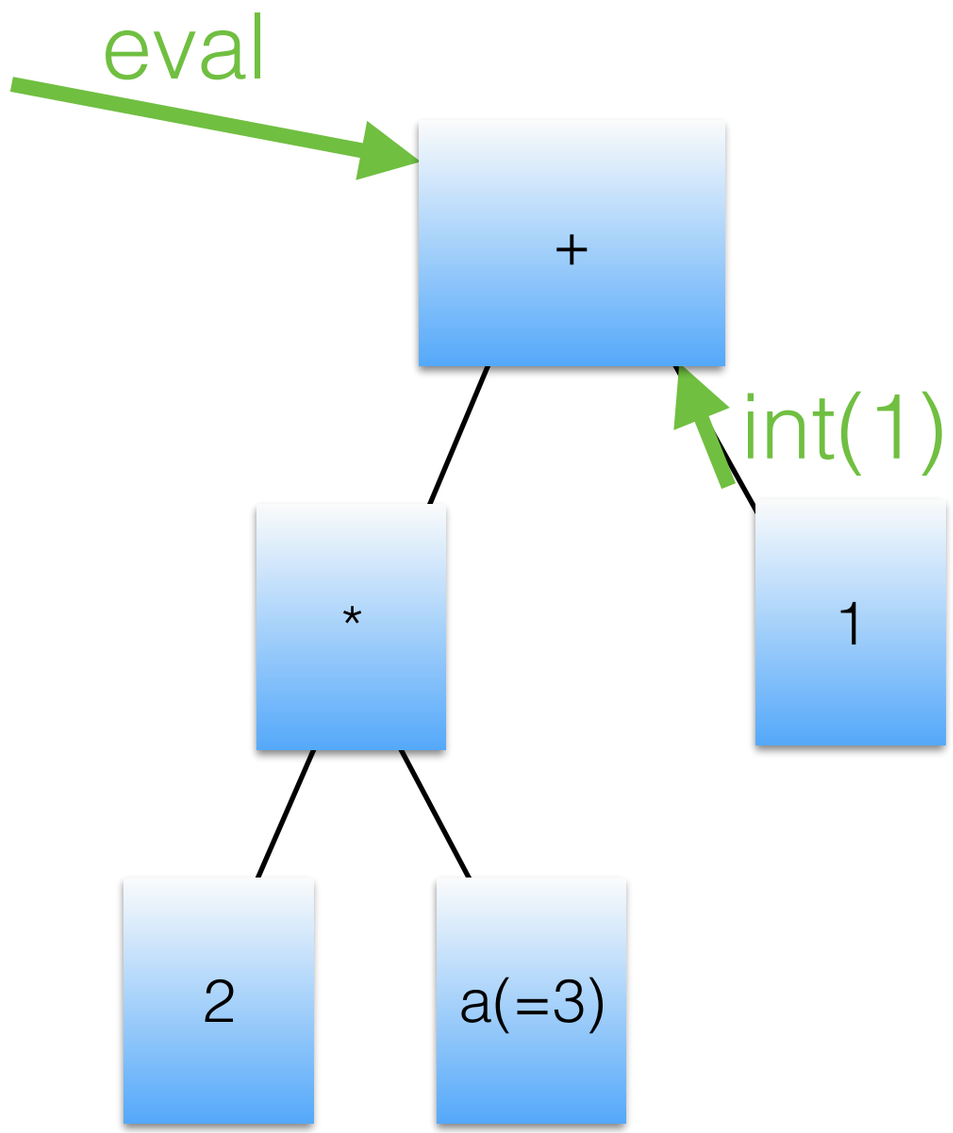
Evolution of an expression
2. Repeated execution



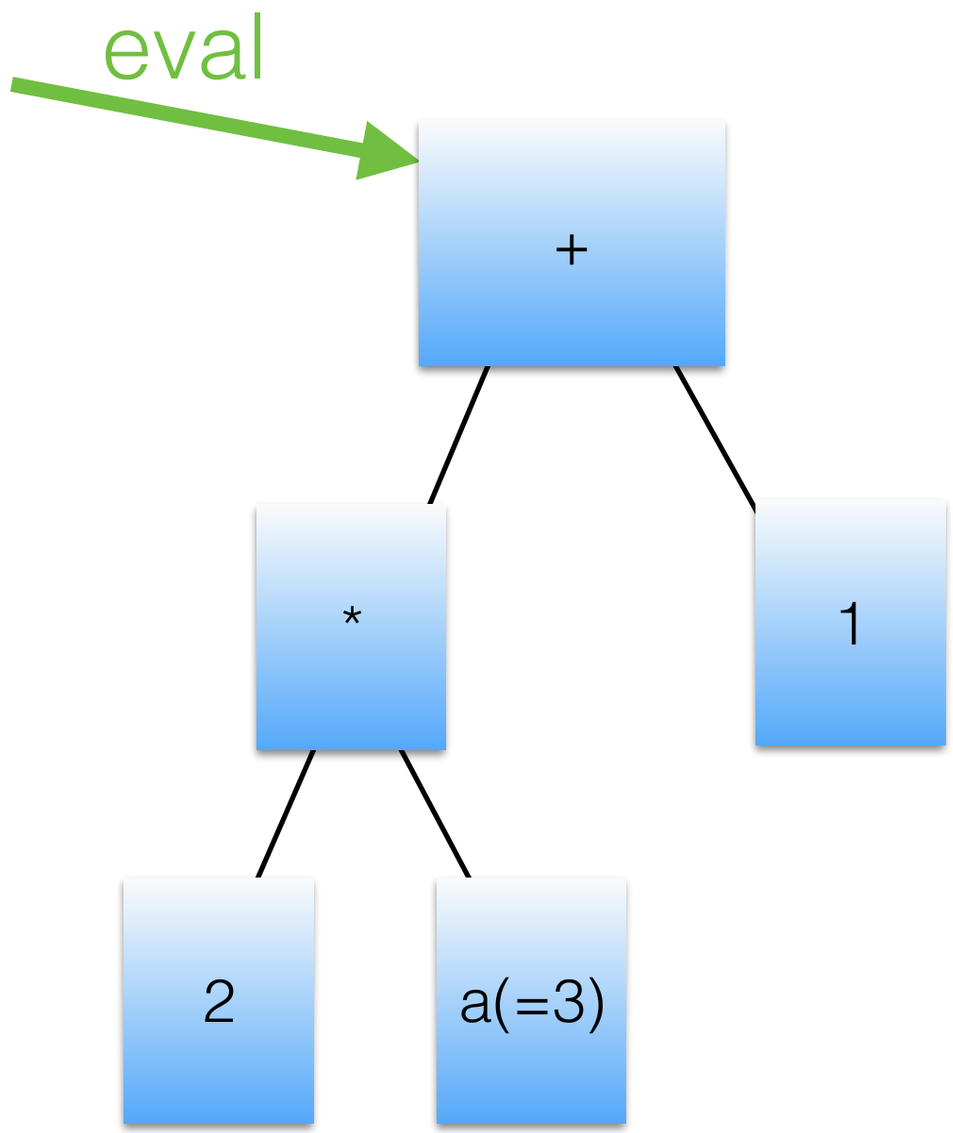
Evolution of an expression
2. Repeated execution



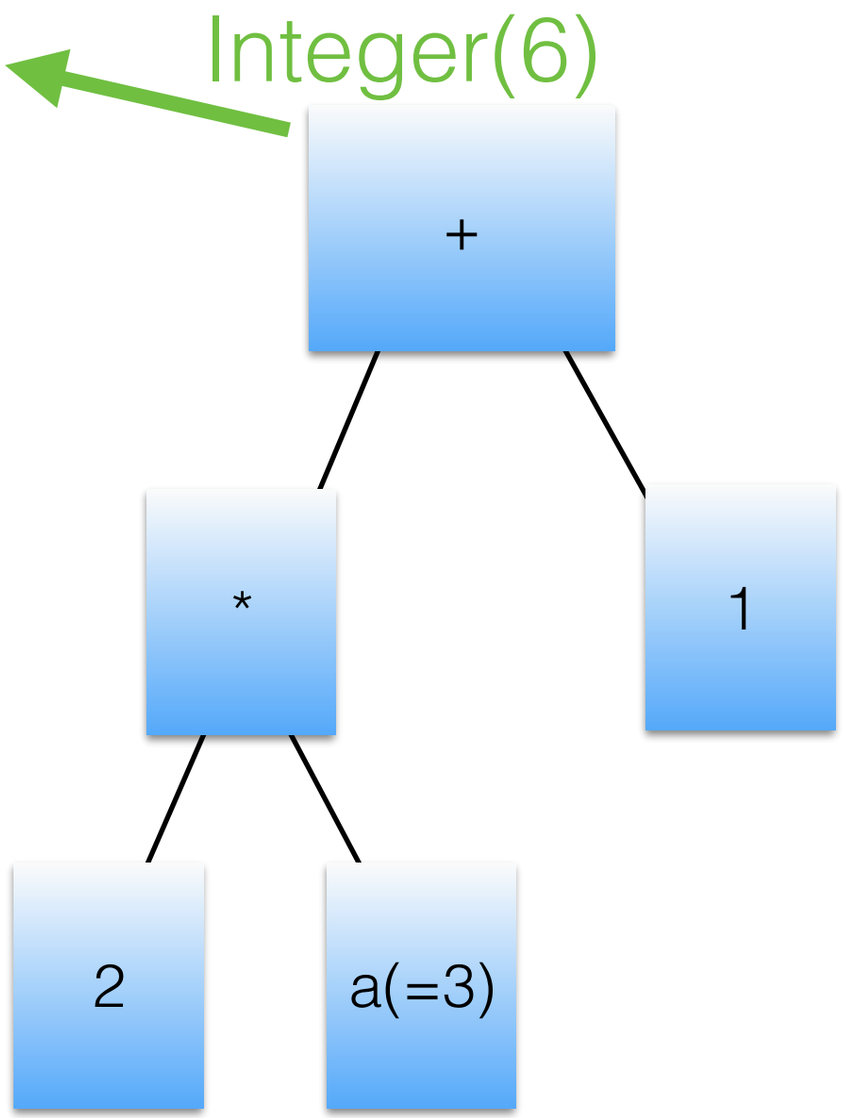
Evolution of an expression
2. Repeated execution



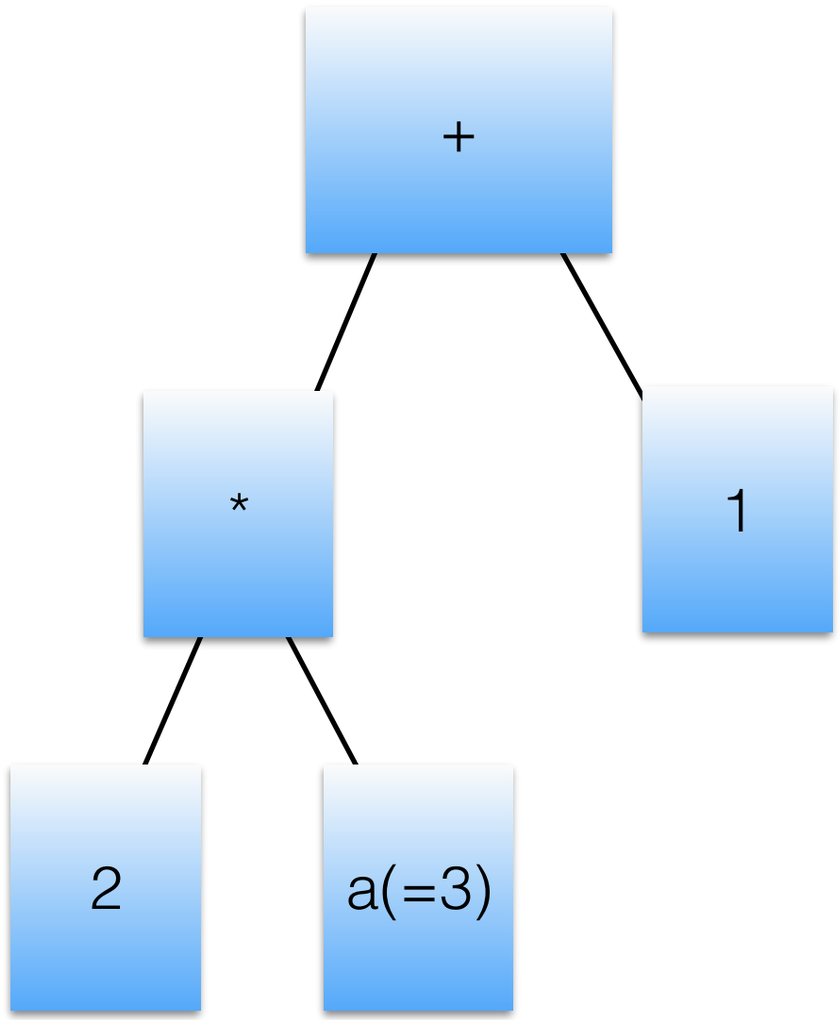
Evolution of an expression
2. Repeated execution



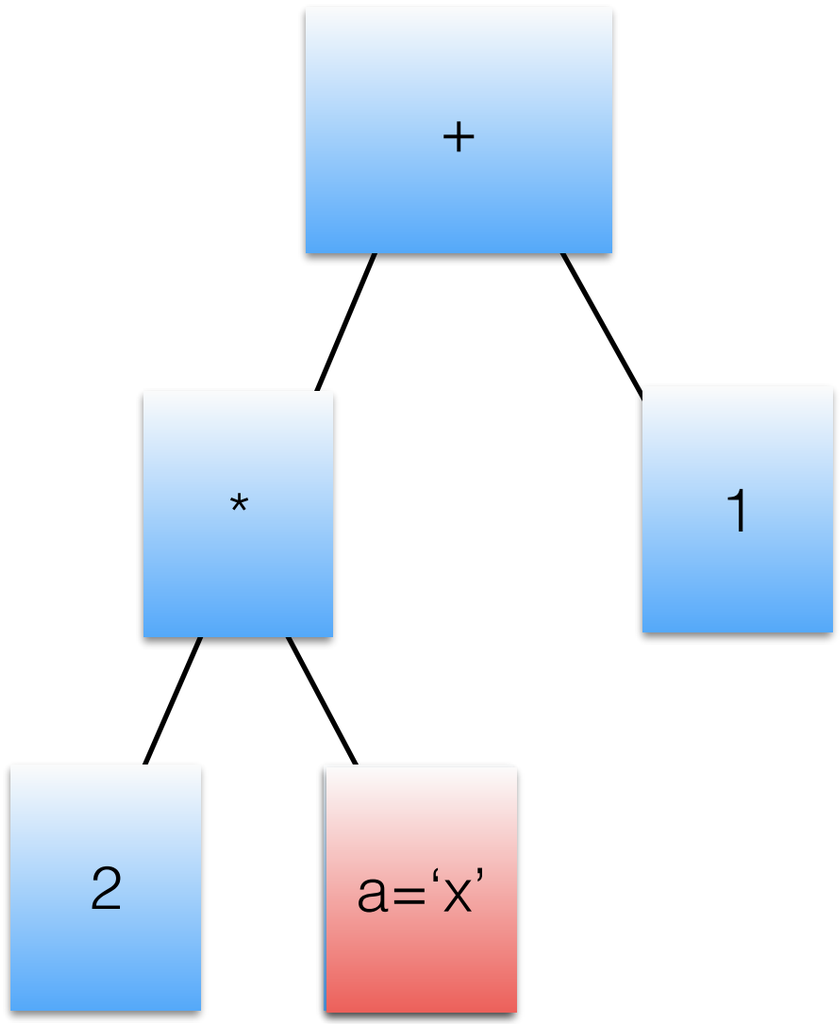
Evolution of an expression
2. Repeated execution



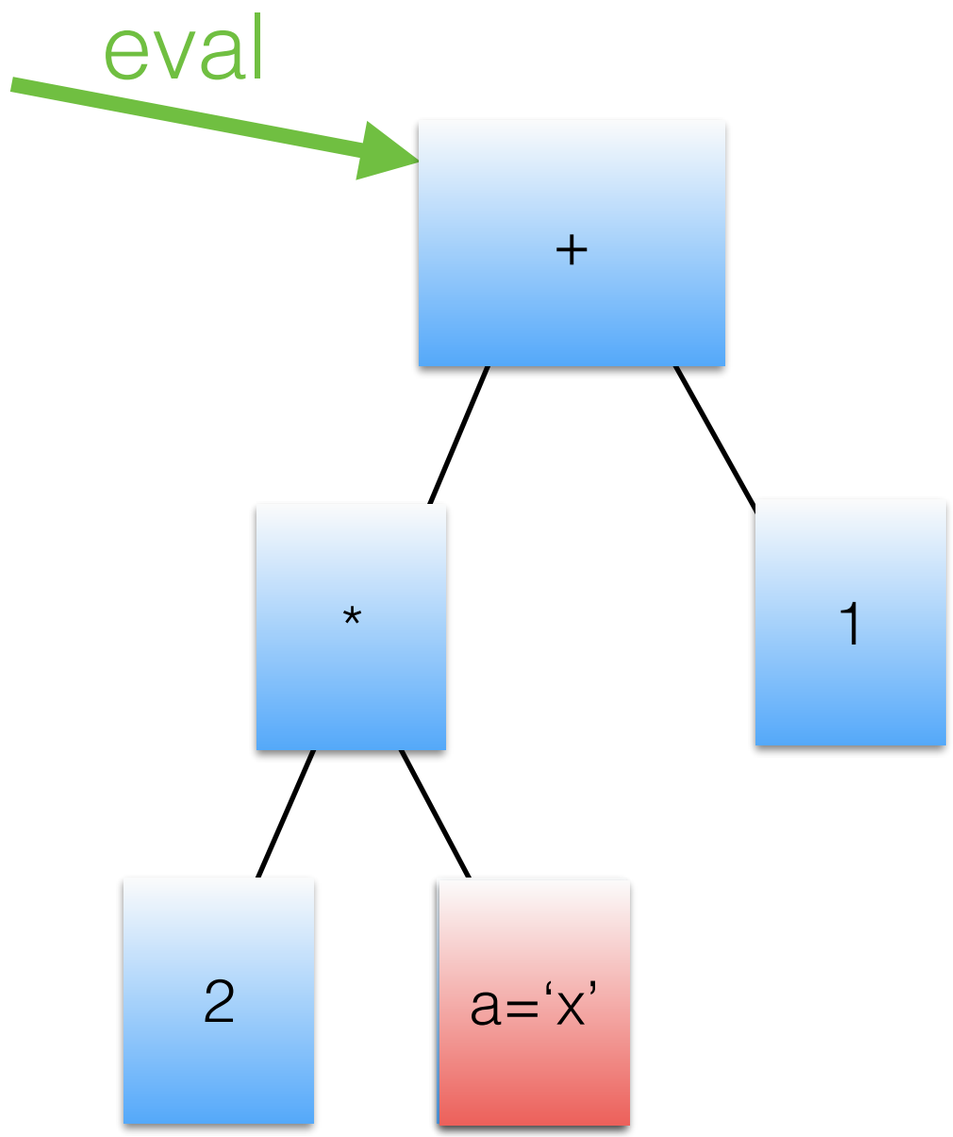
Evolution of an expression
3. Mismatch



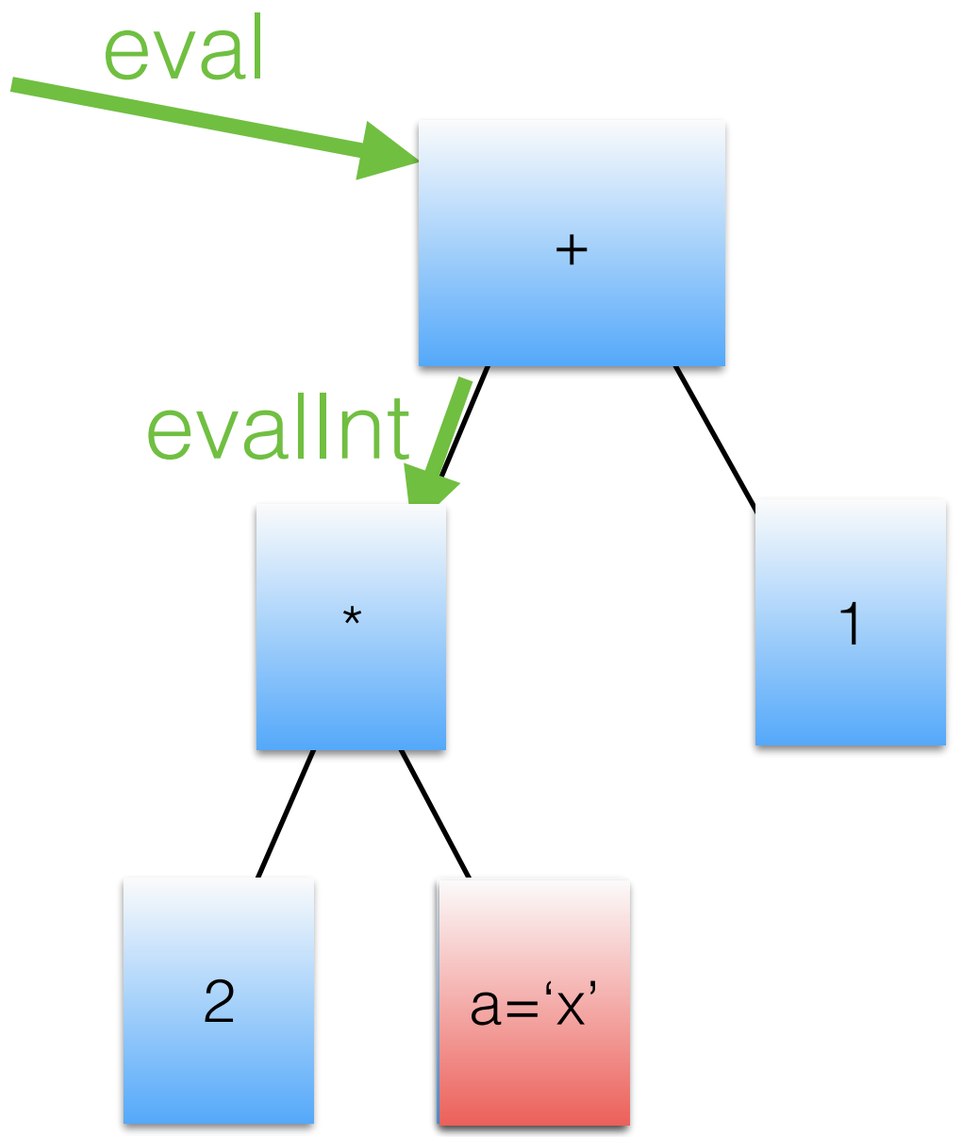
Evolution of an expression
3. Mismatch



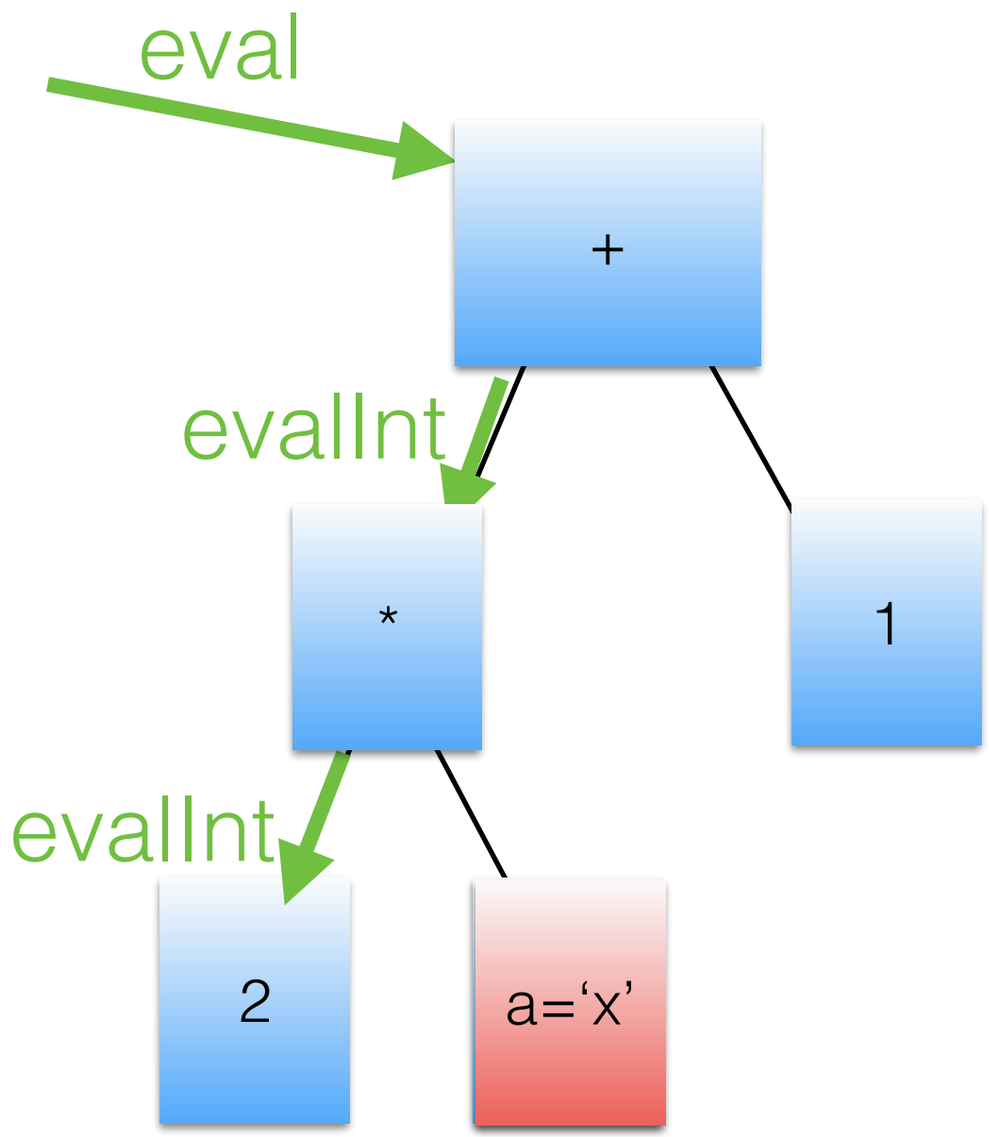
Evolution of an expression
3. Mismatch



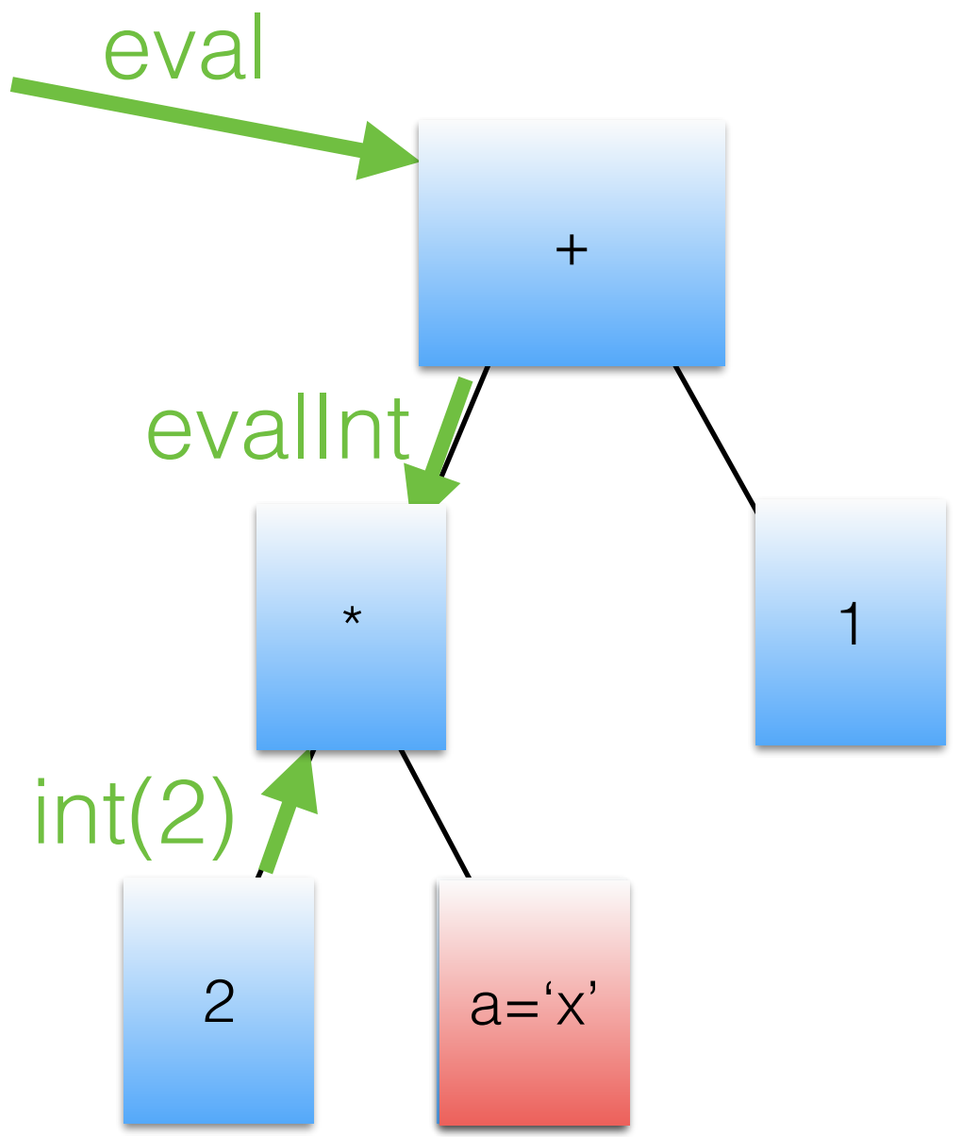
Evolution of an expression
3. Mismatch



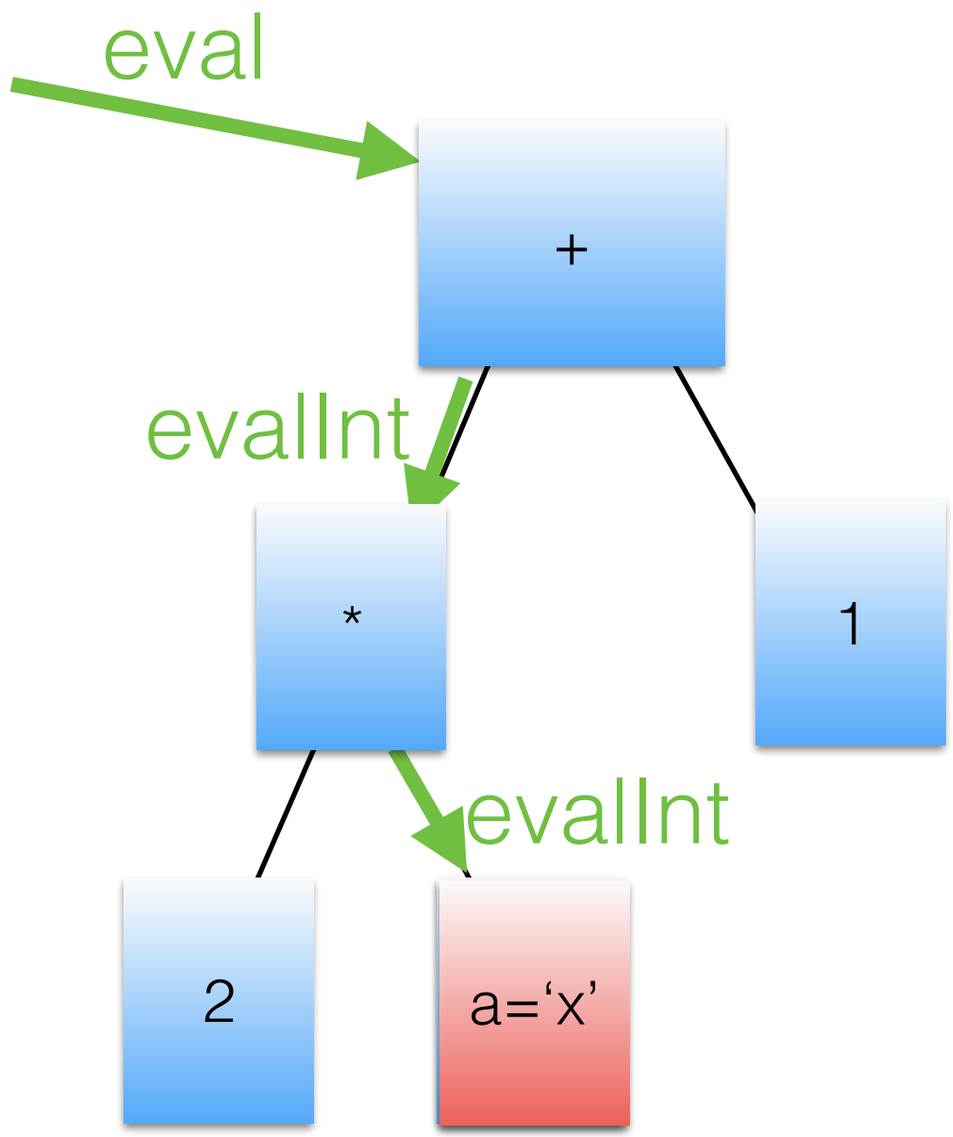
Evolution of an expression
3. Mismatch



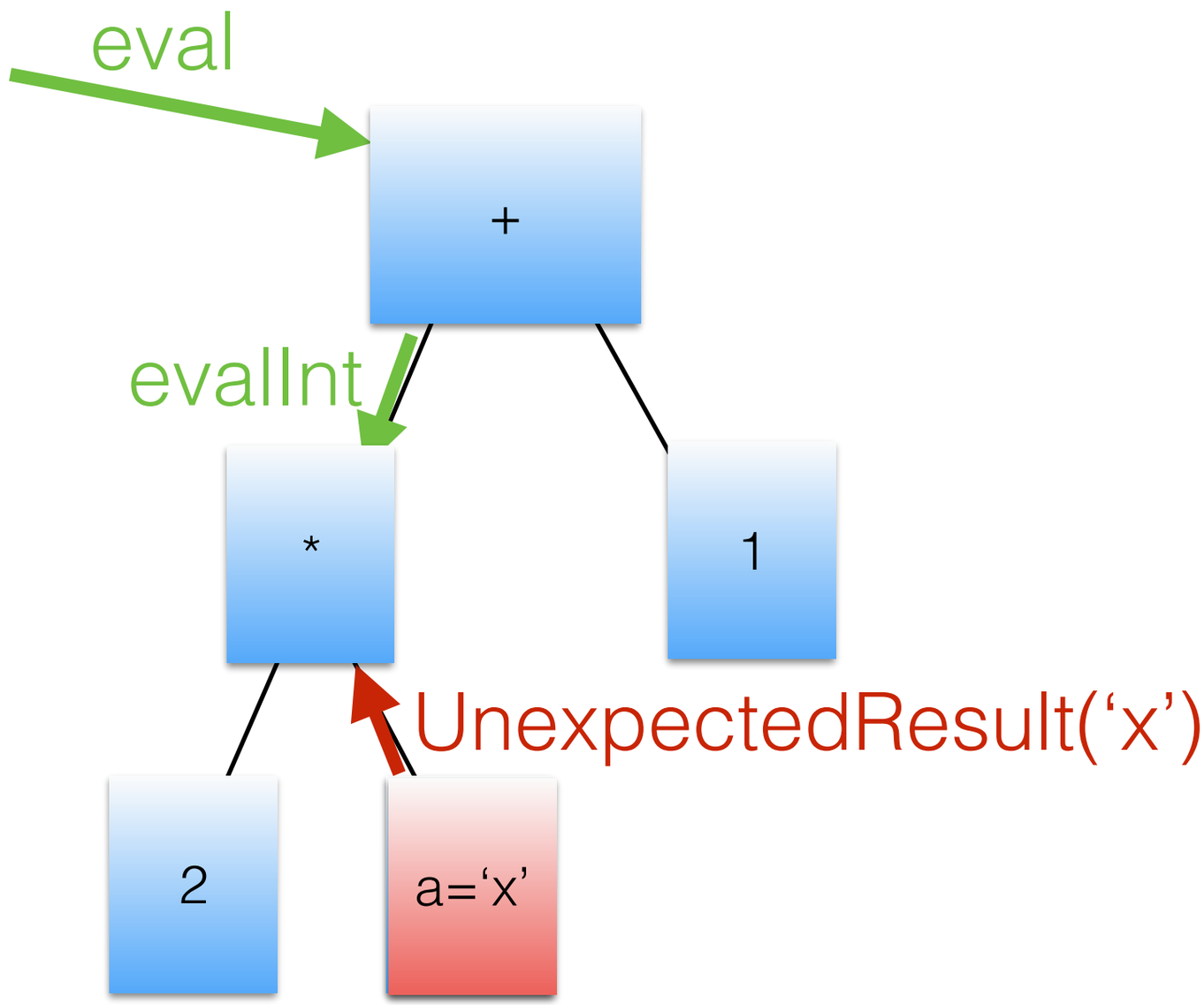
Evolution of an expression
3. Mismatch



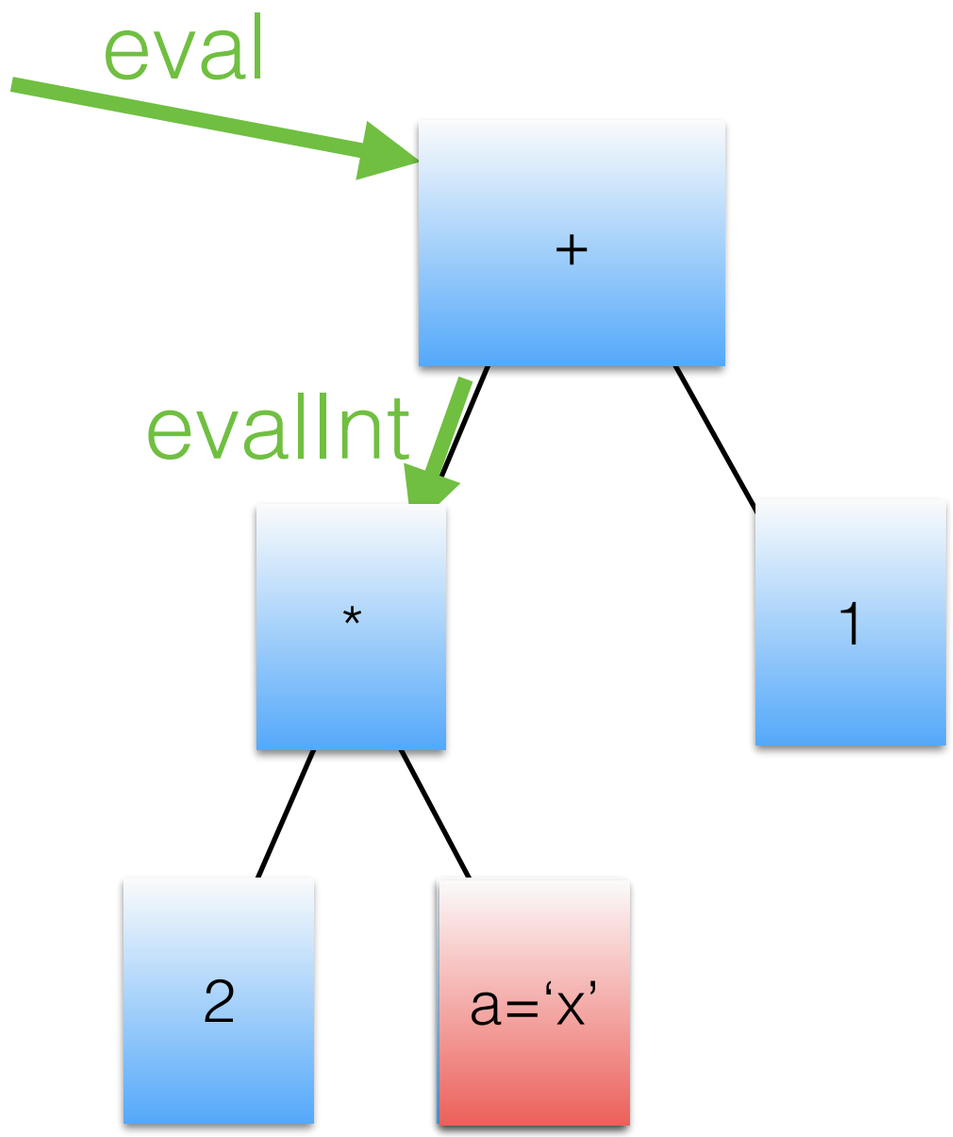
Evolution of an expression
3. Mismatch



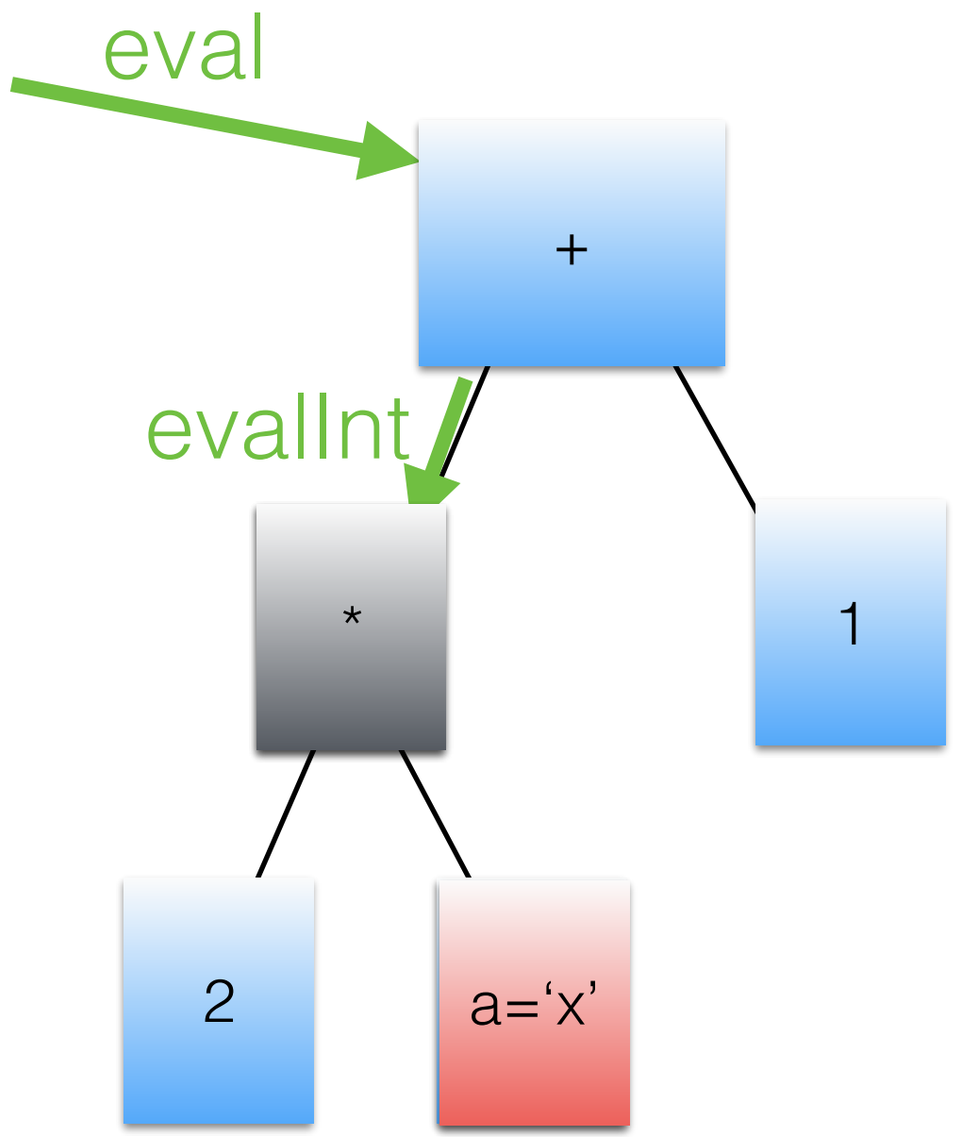
Evolution of an expression
3. Mismatch



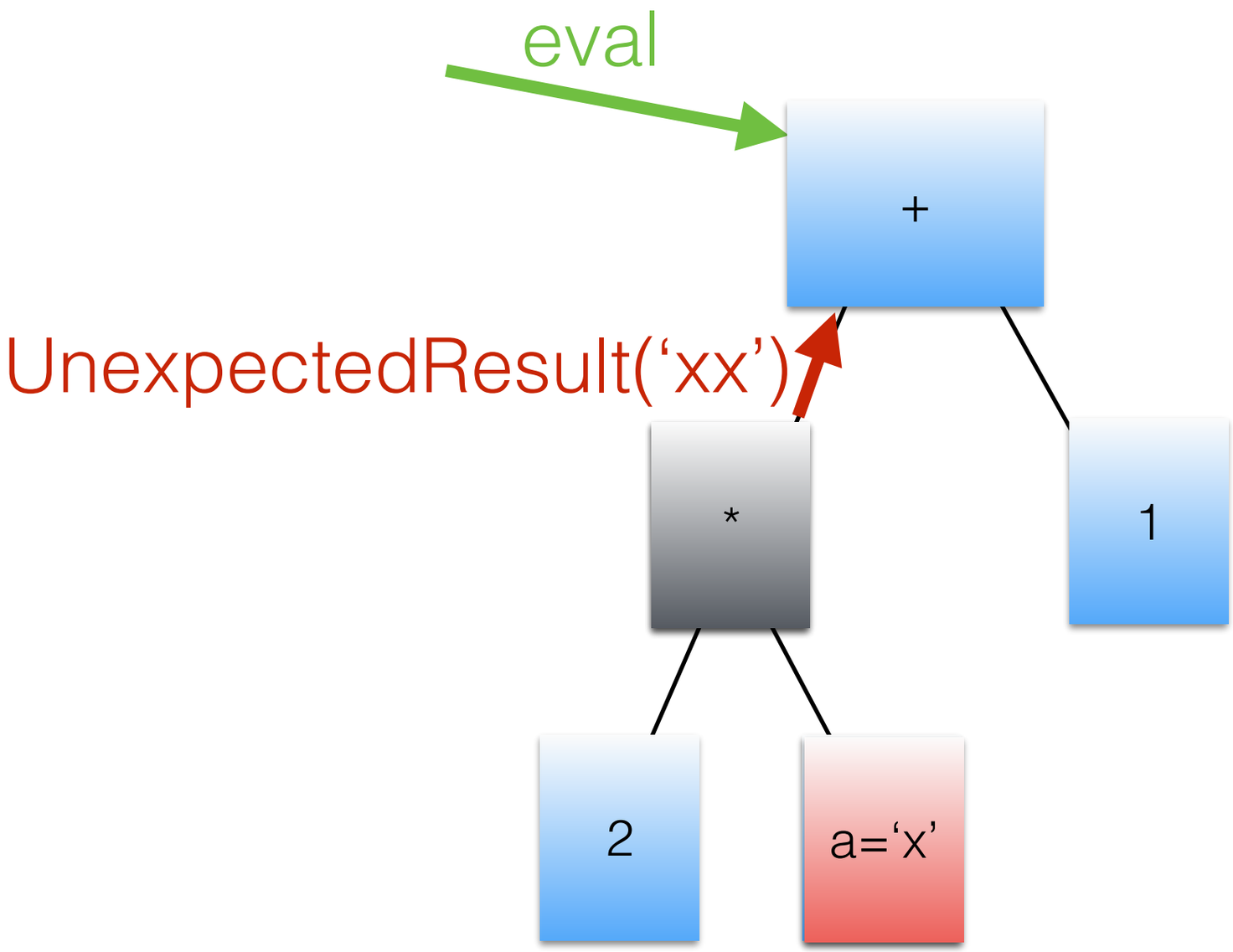
Evolution of an expression
3. Mismatch



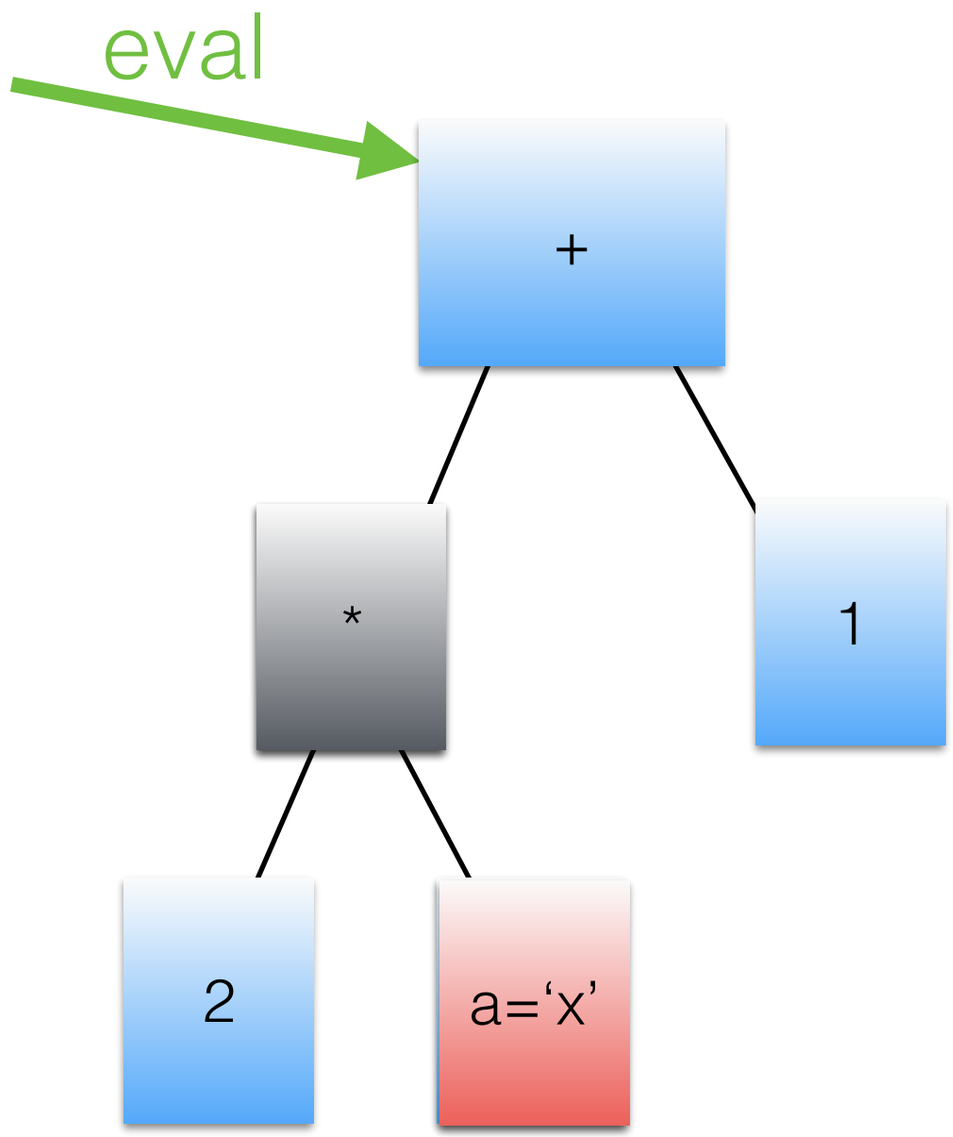
Evolution of an expression
3. Mismatch



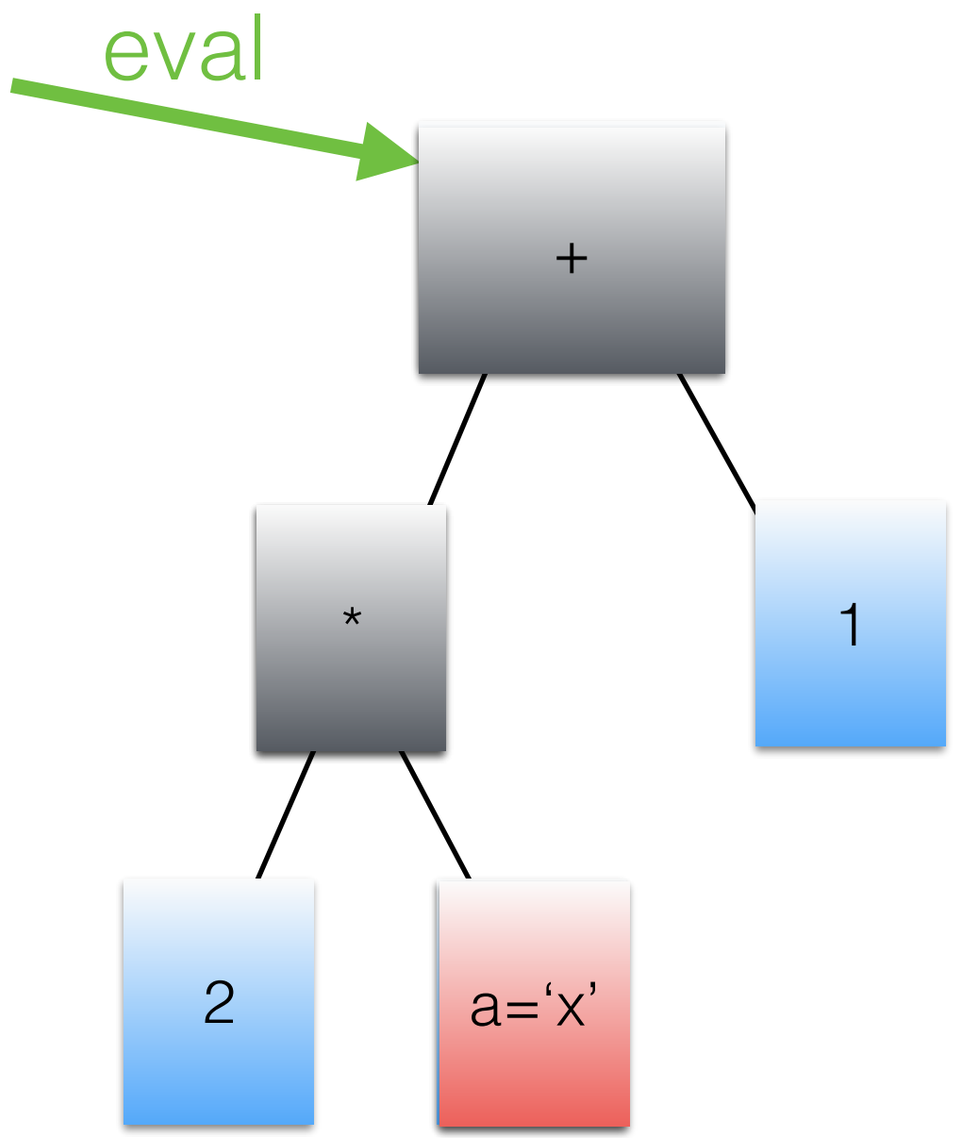
Evolution of an expression
3. Mismatch



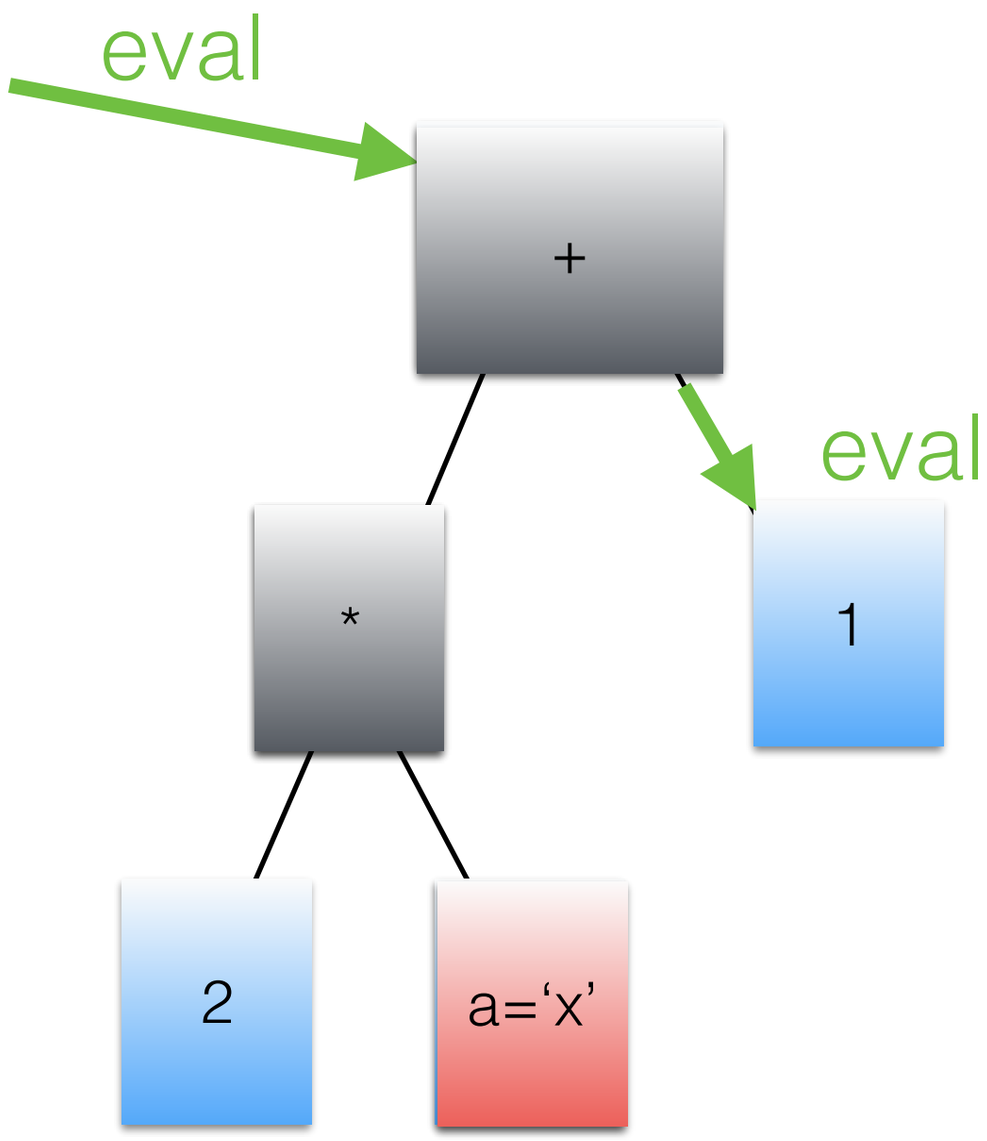
Evolution of an expression
3. Mismatch



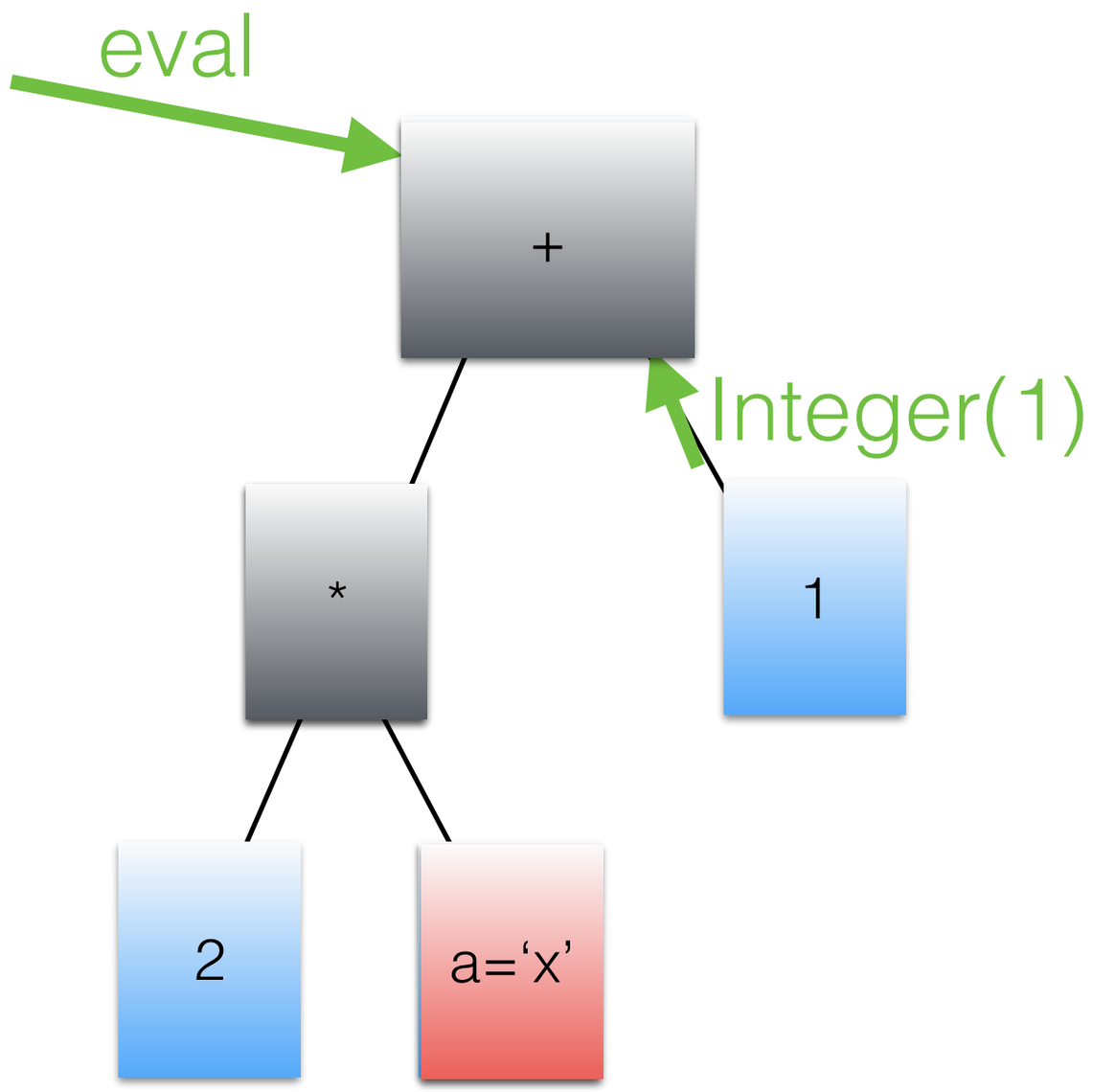
Evolution of an expression
3. Mismatch



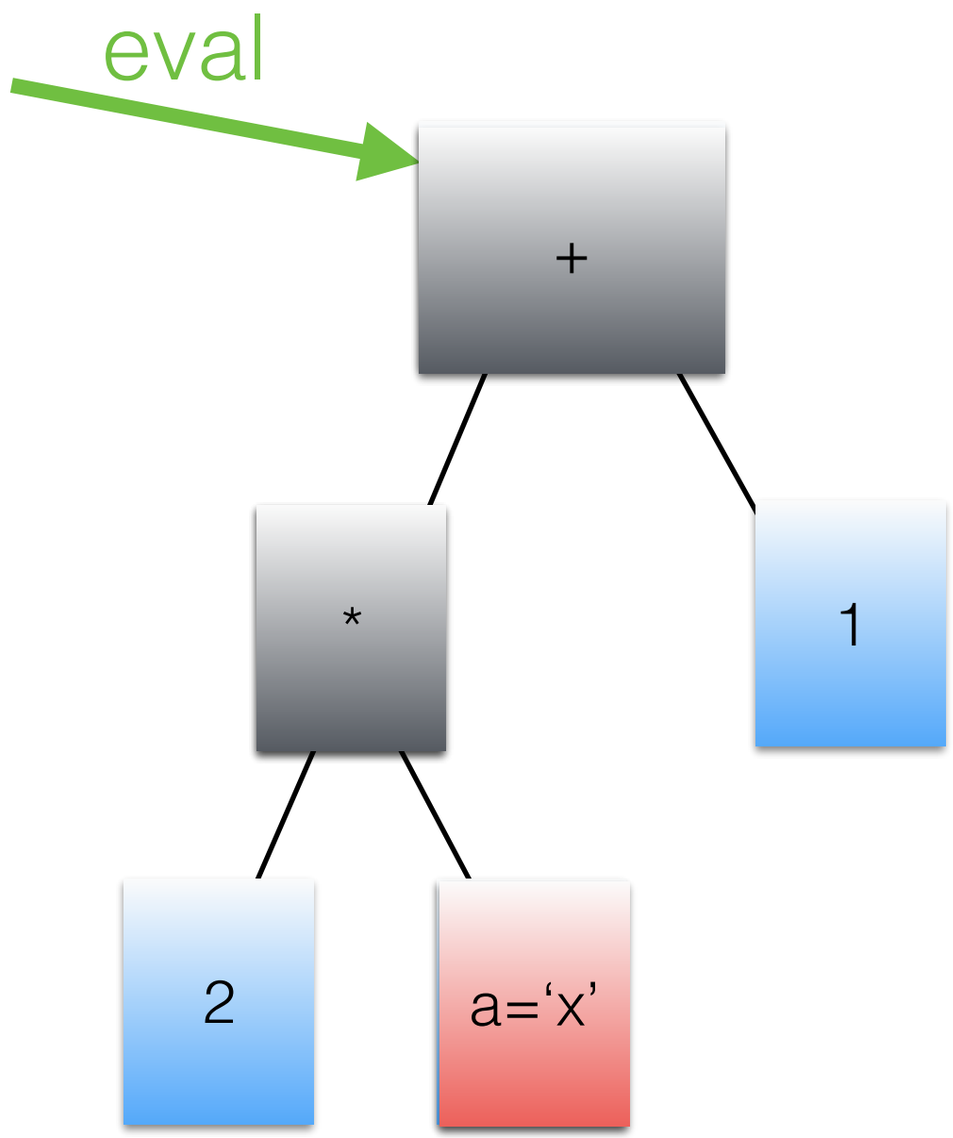
Evolution of an expression
3. Mismatch



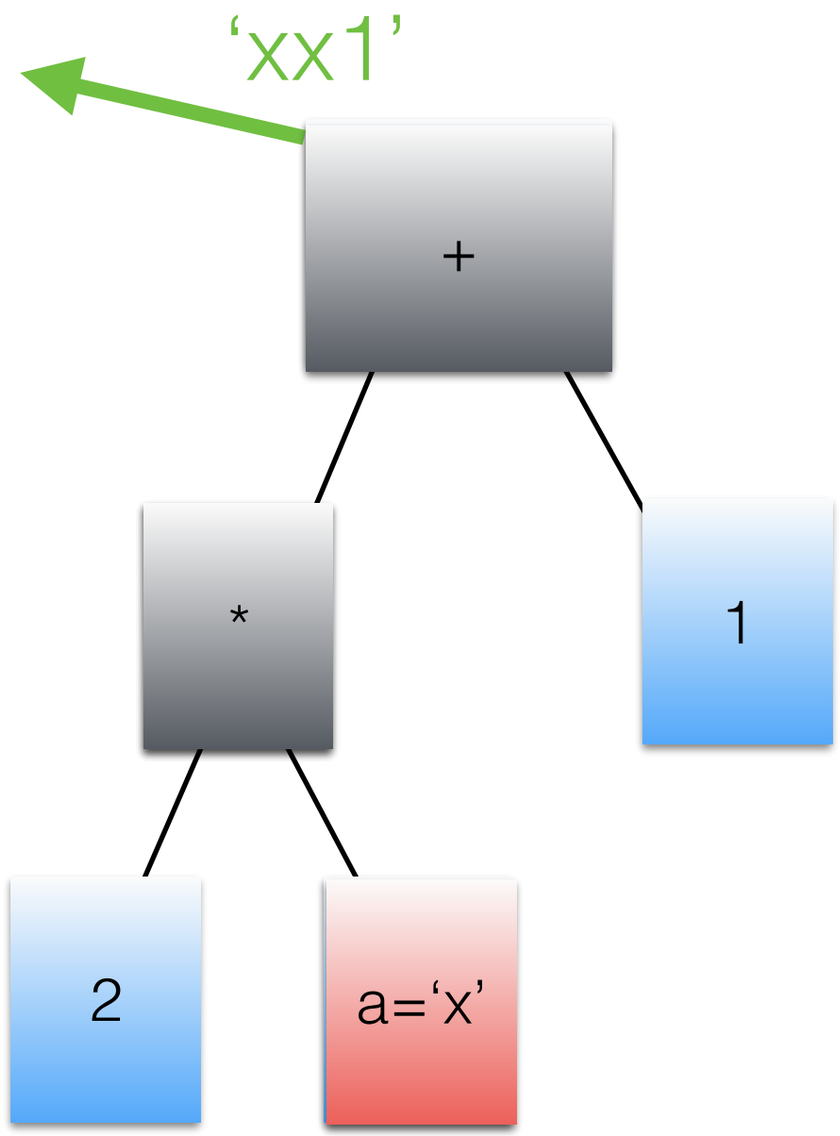
Evolution of an expression
3. Mismatch



Evolution of an expression
3. Mismatch



Evolution of an expression
3. Mismatch

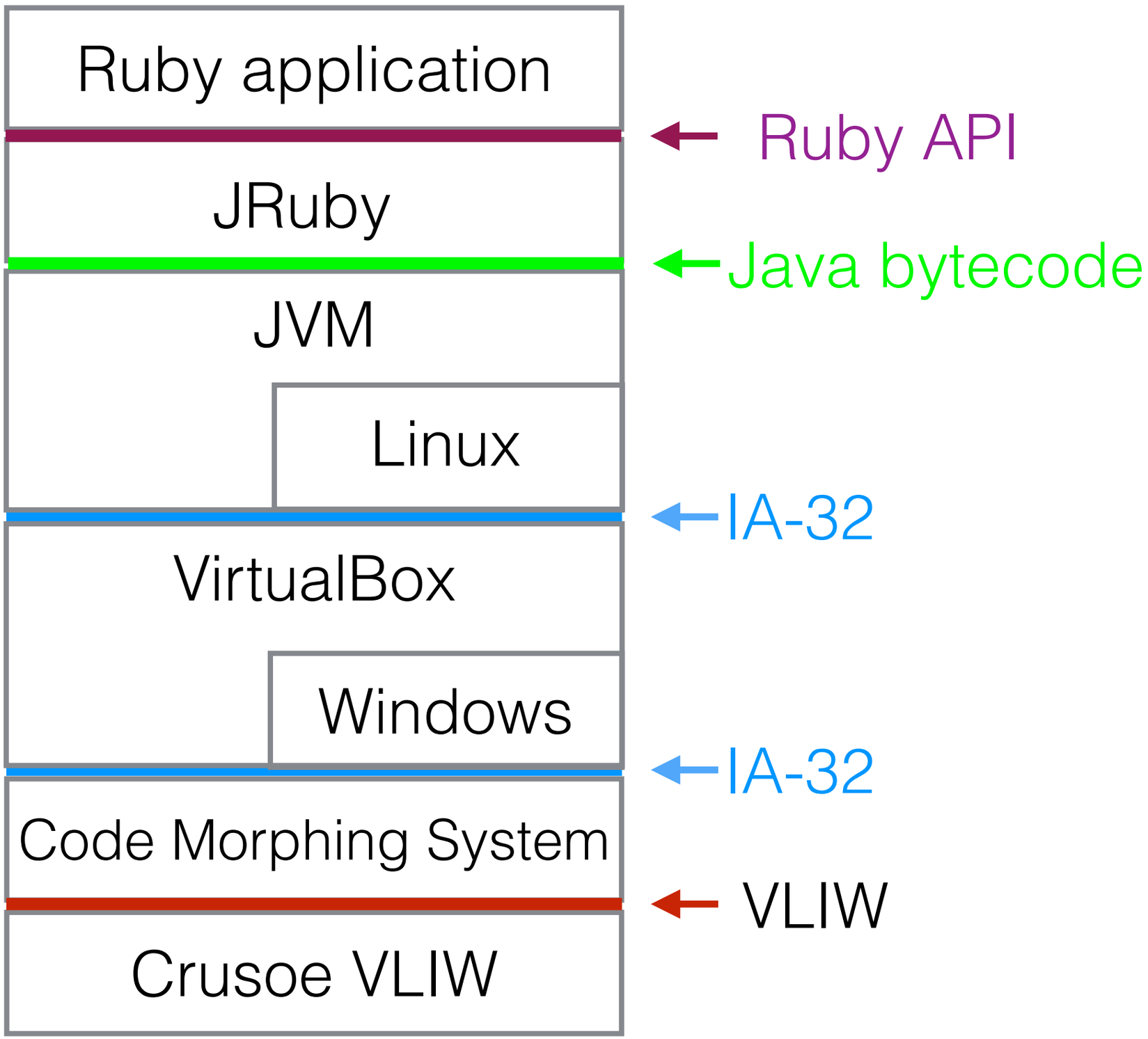


Wrap-up

Smith & Nair VM taxonomy

	Process VMs	System VMs
Same ISA	Dynamic binary optimizers	Classic System VMs Hosted VMs
Different ISA	Dynamic translators Language VMs	Whole-System VMs Co-designed VMs

VMs can be stacked



Why use a VM?

- Portability — decouples the guest from the host.
- Improved security, e.g., via a “sandbox”
- Virtualization of hardware (one piece of hardware can act as many):
    - Consolidation; version management; partitioning of resources; ability to run different flavors of OS simultaneously
- Persistence via snapshots, (live) migration
- Instrumentation/observation
- Convenience: cost-saving and time-saving
    - Don’t have to procure a new system; easier to code for
- Performance — sometimes — with less effort

Confluence of system and language VM technologies

- System VMs and Language VM existed in isolation until the early 1990s.

- Then, dynamic code generation techniques, adopted by language VMs in the 1980s, moved into system VMs.

- Around 2005, trace compilation moved from system VMs to language VMs

The Future

- Multi-lingual support is coming
- Further proliferation as VMs get easier to build
  - New languages? New DSLs?
  - VM scaling vs. hardware node scaling? (Scale-out vs. scale-up)
  - The constant tension between system virtualization and a single OS providing virtual services will always exist (witness Docker).

Acknowledgments

- Thanks to the following for comments on drafts and discussions:
Laurie Tratt, Carl Friedrich Bolz, Peter Kessler, Michael Van De Vanter, Adam Welc, Laurent Daynès, Christian Wimmer.